

Foreword

Logic studies the way we draw conclusions and express ourselves, and deals with how to formalise it. In Logic, propositions are elementary statement and conclusion units; they are analysed both as to the *form*, namely their *syntax*, and the *interpretation*, namely their *semantics*. The relation between the syntax and the semantics is also examined.

The first steps in Logic are credited to the Ionian and Elean philosophers and to the sophists. It is however Aristotle who is considered as the founder of Logic as a science. Somehow, interest in Logic declined with the Roman prevalence in the Mediterranean Sea. And in the Middle Ages, as most of the work of the ancient philosophers — except for Plato and Aristotle — was already lost or had disappeared, Aristotle's syllogistics became the privilege of only a few monks.

Logic regained its interest as non-Euclidian geometries were discovered and as the need for a theoretical foundation of analysis became evident. As soon as 1879, Frege presented the first formal language for Mathematics and Logic. But it was the *paradoxes* of set theory, and the many conversations and disputes among mathematicians of that period on that subject, that gave Logic its final impulsion.

In 1895 and 1897, Cantor published the first and the second part of his study on cardinal and ordinal numbers. In this study, which constitutes the first foundation of set theory, each collection of objects is regarded as a separate entity called a set. Buralli-Forti discovered a paradox in Cantor's theory as soon as 1897. And in 1899, Cantor himself published a paradox concerning his theory on cardinal numbers. Russell published in 1902 a simpler form of that same paradox, known in the history of mathematics as *Russell's paradox*:

In Cantor's theory, each set is defined by the characteristic property of its elements. Let A be the set of all sets X defined by property $X \notin X$,

$$A = \{X \mid X \notin X\}$$

But then:

$$A \in A \text{ if and only if } A \notin A$$

which is obviously an antinomy.

VISIT...

LANZAROTE
Caliente.COM

The paradoxes of set theory generated a great deal of doubt and concern among mathematicians of that period about the well-founding of mathematics. They also made clear that only rigorous formalising of mathematical concepts and method could lead to “sound” theories without antinomies. And so, under Hilbert’s influence, Remark 1.4.1 (2), the axiomatic method’s development began and Logic started taking its actual form.

With the use and the development of computers in the beginning of the 1950’s, it soon became clear that computers could be used not only for arithmetical computation but also for symbolic computation. Hence, the first arithmetical computation programs, and the first programs created to answer elementary questions and prove simple theorems, were written simultaneously. The basic steps towards a general method based on Logic, which would allow statement procedures as well as manipulation of symbols, were first accomplished in 1965 by Robinson with the use of the *resolution method*, and later by Kowalski and Colmerauer who made use of Logic directly as a Logic Programming language.

This book unfolds Logic’s basic elements; it analyses the relation of and the transition from Logic to Logic Programming.

The first chapter describes *Propositional Logic* (PL). It examines, in the first place, the propositional language, as well as the syntax and the semantics of PL formulae. Three proof methods are presented afterwards: the axiomatic proof method, mainly for historical reasons, the Beth tableaux method, with proofs using mathematical analysis of propositions, and the resolution method. The last method uses proofs in a similar but much more effective formalism that can easily be materialised in a program. These proof methods are validated by corresponding soundness and completeness theorems.

The second chapter examines *Predicate Logic* (PrL). In Predicate Logic, structures of propositions are more extensively analysed, for the predicate language is much richer than the propositional language. The predicate language contains indeed quantifiers and constants, as well as predicates and functions. PrL semantics are hence much more complex. Herbrand interpretations are of great importance to these semantics, because they make satisfiability tests for PrL propositions possible by using PL methods. The axiomatic proof method, the Beth tableaux method as well as the resolution method, which by means of the unification algorithm constitutes the basis of Logic Programming, are examined afterwards. Corresponding soundness and completeness theorems confirm the validity of these proof methods.

The third chapter deals with *Logic Programming* as well as with the programming language PROLOG, and it also analyses PROLOG's derivation mechanism. Logic supplies Logic Programming and PROLOG with both a very rich language and the theoretical foundation which guarantees correct results.

Each chapter includes solved as well as unsolved exercises provided to help the reader assimilate the corresponding topics. The solved exercises demonstrate how to work methodically, whereas the unsolved exercises aim to stimulate the reader's personal initiative. The contents of this book are self-contained, in the sense that only elementary knowledge of Analysis is required to study them. This book can therefore be used by students in every academic year, as a simple reading, or in the context of a course. It can also be used by those who utilize Logic Programming without having any particular theoretical background knowledge of Logic, and by those simply interested in Logic and its applications in Logic Programming. All topics developed in this book are constantly analysed within the perspective of a transition *from Logic to Logic Programming*. Therefore, certain proofs, mainly those of completeness of the axiomatic method proofs, are omitted. References to the international bibliography direct to more complete developments of concepts and proofs that are only indirectly related to the perspective of the book, and therefore not extensively examined.

The material in this book is based on the book published by the first author in Greek in 1992 and which, in turn, drew on the authors' seminars on the Principles of Logic Programming, as well as on teaching at the universities of Cornell and Rochester in the United States, and at the University of Patras, in Greece.

At this point, the contributions of Aneta Sinachopoulos-Svarna in sections 1.1, 1.8, 2.3, 2.11, 3.2, 3.3, 3.6, 3.8, 3.9 and in the material's general configuration and presentation, and of George Potamias in units 3.4, 3.5 and 3.7 must be stressed. Costas Kontogiannis, Spiros Papadakis and Petros Petropoulos, postgraduate students of the university of Patras, also contributed to the material's preparation. All programs included in the text and in the exercises have been run by George Potamias in TURBO-PROLOG, v.2.0.

George Metakides
Anil Nerode

I Propositional Logic

Μέτρον πάντων χρημάτων ἄνθρωπος·
τῶν μὲν ὄντων ὡς ἐστί, τῶν δὲ μὴ ὄντων
ὡς οὐκ ἐστί.

Man is the measure of all things; of the things
that are, that they are; of the things that are
not, that they are not.

Protagoras

1.1 Introduction

Until 1850, mathematical proofs were based on experience and intuition and were generally accepted. Leibniz was the first to outline, as early as 1666, the need of a formal language, called “universal characteristic”, which would be created and used in mathematical formulations and methodology. However his ideas were ahead of their time. The initial development of a formalism for mathematics is credited to Boole, two hundred years later, with his studies on “The Mathematical Analysis of Logic” published in 1847, and “The Laws of Thought” published in 1854. De Morgan also contributed to the efforts for the creation of a formalism with publications of his own in 1847 and 1864, as well as Peirce in 1867 and Frege in 1879, 1893, and 1903. The publication of Russell and Whitehead’s three-volumed “Principia Mathematica” (Mathematical Principles) marked the acme of these efforts, for it presented a logical system within which all mathematics was formalized and all corresponding theorems were derived by means of logical axioms and rules.

Logic flourished as an independent science after 1920, with Lukasiewicz (1878-1956), Lewis (1883-1964), Gödel (1906-1978), Tarski (1901-1983), Church (1903-1995) and Kleene (1909-1994) as its main representatives.

Starting in the nineteen fifties, computer technology dealt with how to utilise a computer for symbolic computation. A solution to that problem was provided by Functional Programming, created by McCarthy among others, and used mainly in the United States. Another solution was provided by Logic Programming, created mainly in Europe by Colmerauer and Kowalski among others. Logic Programming gave Logic a new impulse: it solved many problems concerning forms of logical propositions which should allow the computer to execute syllogisms and judgement procedures. Logic Programming also brought answers to questions concerning the nature and the mechanisms of logical procedures executed by the computer.

We thus need to examine, in the first place, how mathematical and logical propositions are formalised. Logical connectives are the basic elements of such a formalisation, as we can see in the following examples.

- (1) Conjunction is formalized by \wedge . Let us assume that we know two properties of a certain x :

$$A : x > 3$$

$$B : x < 10$$

Then we know about x that it is greater than 3 and that it is smaller than 10. In other words, we know the proposition:

$$A \wedge B$$

where the logical connective \wedge corresponds to the grammatical conjunction “and”. Proposition $A \wedge B$ thus states that “ $x > 3$ and $x < 10$ ” which means “ $3 < x < 10$ ”.

- (2) Negation is formalised by means of the symbol \neg . For example, consider the proposition

$$C : 50 \text{ is divisible by } 7$$

$\neg C$ denotes that “50 is not divisible by 7”

- (3) Disjunction is denoted by the symbol \vee . If D and E are the propositions

$$D : 60 \text{ is a multiple of } 6$$

$$E : 60 \text{ is a multiple of } 5$$

then the proposition

$$D \vee E$$

states that “60 is a multiple of 6 or 60 is a multiple of 5”. The symbol \vee is not exclusive, since 60 is a multiple of both 6 and 5, as well as a multiple of 20 which is not mentioned in propositions D and E .

- (4) The implication “if ... then ... ” is denoted in Logic by “ \rightarrow ”. If F and G are the propositions

$$\begin{aligned} F : & \text{ the number } a \text{ is a multiple of 10} \\ G : & \text{ the number } a \text{ is a multiple of 5} \end{aligned}$$

then the proposition

$$F \rightarrow G$$

states that “if a number a is a multiple of 10, then it is a multiple of 5”.

- (5) “If ... and only if ... ” is denoted by the equivalence symbol “ \leftrightarrow ”. For example, if H and I are the propositions

$$\begin{aligned} H : & \text{ 16 is a multiple of 2} \\ I : & \text{ 16 is an even number} \end{aligned}$$

then formally we can write:

$$H \leftrightarrow I$$

Logical connectives do not all have the same properties. In the example of propositions A and B , writing $A \vee B$ does not differ conceptually from writing $B \vee A$. Thus intuitively, the logical connective \vee seems to be commutative, and the same holds for \wedge . However the question whether the connective \rightarrow is commutative cannot be answered positively: the proposition $G \rightarrow F$, “if a is a multiple of 5, then a is multiple of 10”, does not seem “right”, since 15 is a multiple of 5 but no multiple of 10. Therefore we need to study the properties of logical connectives.

There are also several kinds of propositions. For instance, propositions such as:

“6 is a multiple of 6”

“7 is a prime number”

“a rational number is 0 or $\neq 0$ ”

which are “good”, “correct” propositions,

or propositions such as

“50 is an even number”

“50 is divisible by 7”

“if $x = 3$, then $x = 5$ ”

which are “bad”, “erroneous” propositions,

or propositions such as

“It’s raining”

“I’m hungry”

“ x is equal to its absolute value”

which are sometimes “correct” and sometimes “erroneous”.

Intuitively, each proposition has a certain interpretation: it can express truth or falsehood, that is to say, it may be true or false. But what makes a proposition true? What is the relation between the interpretation of a compound proposition and the interpretations of the propositions which constitute it? And what role do logical connectives play in the interpretations? We need to study interpretations of propositions and determine the role of logical connectives in these interpretations.

Our own logic allows us to make decisions and draw conclusions. For instance, we say “if $x > 3$ then $x > 0$ ”, or formally:

$$(x > 3) \rightarrow (x > 0)$$

Let us assume that $x > 3$. Then $x > 0$. Which rule allows us to conclude “ $x > 0$ ” when the assumption “ $x > 3$ ” is valid? And does this rule have a general validity; in other words, if $M \rightarrow N$ is valid and M is valid, is proposition N always valid?

Here is another example. We know that in order to go to the movies we need money for our ticket.

$$\text{movies} \rightarrow \text{money for the ticket} \tag{1}$$

Let us assume that we do not have enough money for the ticket:

$$\neg (\text{money for the ticket}) \tag{2}$$

Are we able to go to the movies? What is the impact of negation? What form will (1) take with the negation?

$$\neg (\text{movies}) \rightarrow \neg (\text{money for the ticket}) ?$$

Or $\neg (\text{movies} \rightarrow \text{money for the ticket}) ?$

Or $\neg (\text{money for the ticket}) \rightarrow \neg (\text{movies}) ?$

And what comes as a valid conclusion when (1) and (2) are valid?

We therefore need to examine the syntax of propositions, and the rules that determine when and how we can deduce valid conclusions from premises from available data. Propositional Logic, as well as Predicate Logic, which is more complex, deals with these topics.

In the Propositional Logic (PL) chapter, we will examine PL propositions with respect to both their syntax and semantics. We will also study methods which deduce conclusions from premises and we will determine the adequate forms of PL propositions for knowledge representation, decision procedures and automatic theorem proving in Logic Programming.

1.2 The Language of Propositional Logic

Alphabet, Syntax and Semantics

The language of Propositional Logic (PL) is a formal language. By means of this language, we formalize that specific part of everyday speech which is necessary to the formulation of logical and mathematical concepts. Having formalized propositions with the use of that language, we next examine them according to their structure and validity.

For every formal language, we have:

- (a) an **alphabet** containing all the symbols of the language
- (b) a **syntax** by means of which the symbols are utilised and the propositions are formed
- (c) **semantics** by means of which the language interprets and allocates meanings to the alphabet symbols.

The alphabet of the language of Propositional Logic consists of:

- (i) Propositional symbols: $A, A_1, A_2, \dots, B, B_1, B_2, \dots$
- (ii) Logical connectives: $\vee, \wedge, \neg, \rightarrow, \leftrightarrow$
- (iii) Commas and parentheses: “ , ” and “ (” , “) ”

Logical connectives intuitively correspond to conjunctions used in everyday speech. We thus have the following correspondence:

\vee ,	disjunction :	or
\wedge ,	conjunction :	and
\rightarrow ,	implication :	if ... then ...
\leftrightarrow ,	equivalence :	... if and only if ...
\neg ,	negation :	not

Any sequence of symbols belonging to the alphabet of a language is called an expression. For instance, $A \vee \vee B$, $A \vee B$, $\leftrightarrow A$ are expressions of the Propositional Logic’s language. Certain **well-formed** expressions are considered by the syntax as propositions. The following definition looks into the concept of propositions and, furthermore, describes the syntax laws of the PL language.

Definition 1.2.1: Inductive definition of propositions:

- (i) The propositional symbols are propositions, called **atomic propositions** or **atoms**.
- (ii) If σ, τ are propositions, then the expressions $(\sigma \wedge \tau)$, $(\sigma \vee \tau)$, $(\sigma \rightarrow \tau)$, $(\sigma \leftrightarrow \tau)$, $(\neg \sigma)$ are also propositions, called **compound propositions**.
- (iii) The expressions constructed according to (i) and (ii) are the only expressions of the language which are propositions. ■ 1.2.1

The expressions $\vee A \vee B$ and $\leftrightarrow A$ are thus not propositions since they were not constructed according to (i) and (ii), whereas $(A \vee B)$ and $((\neg A) \vee (B \leftrightarrow (\neg C)))$ are indeed propositions.

The above definition uses the method of induction to define compound propositions; (i) is the **first step** and (ii) is the **inductive step**. The induction is on the logical length, the structure of the propositions. By the “logical length” of a proposition A , we mean a natural number denoting the number, the kind and

the order of appearance of the logical connectives used in the construction of A , starting from the atomic propositions which constitute A . We do know that the **principle of mathematical induction** holds for the natural numbers:

Definition 1.2.2:

If $P(a)$ denotes a property of a natural number a , and if we have:

- (1) $P(0)$, and
- (2) for any natural number n , if $P(n)$ holds true then we can prove that $P(n + 1)$ holds true,

then we can conclude that for all the natural numbers n , $P(n)$ holds. ■ 1.2.2

The well-ordering of the natural numbers (which means that every non-empty subset of the natural numbers has a least element), together with the principle of induction, leads to a second form of induction, called **complete** or **course of values induction**.

Definition 1.2.3: Complete Induction:

If $P(a)$ is a property of a natural number a , and if we have

- (1) $P(0)$, and
- (2) For all the natural numbers m and n , such that $m < n$, we can derive $P(n)$ from $P(m)$,

then for all the natural numbers n , $P(n)$ holds true. ■ 1.2.3

We used complete induction to give the inductive definition of propositions above (Definition 1.2.1), and will use it again to give the following definition.

Definition 1.2.4: General scheme of induction for propositions:

Let P be any propositional property. We shall write $P(\sigma)$ to mean that **proposition σ has the property P** . If we prove that:

- (a) $P(A)$ holds true for any atom A of the language
- (b) if σ_1, σ_2 are propositions and $P(\sigma_1)$, $P(\sigma_2)$, then $P((\sigma_1 \wedge \sigma_2))$, $P((\sigma_1 \vee \sigma_2))$, $P((\sigma_1 \rightarrow \sigma_2))$, $P((\sigma_1 \leftrightarrow \sigma_2))$, $P((\neg \sigma_1))$ and $P((\neg \sigma_2))$

then we can conclude $P(\sigma)$ for any proposition σ of the language. ■ 1.2.4

Example 1.2.5:

- (i) The property P : “the number of left parentheses is equal to the number of right parentheses” holds true for all propositions of PL.

Indeed, according to the general scheme of induction, we have:

- (a) In the propositional symbols, the number of left and right parentheses is equal to 0.
 - (b) If in the propositions σ_1 and σ_2 , the number of left parentheses is equal to the number of right parentheses — let n and m be these numbers respectively for σ_1 and σ_2 — then in the propositions $(\sigma_1 \wedge \sigma_2)$, $(\sigma_1 \vee \sigma_2)$, $(\sigma_1 \rightarrow \sigma_2)$, $(\sigma_1 \leftrightarrow \sigma_2)$, $(\neg \sigma_1)$ and $(\neg \sigma_2)$ the number of left parentheses is equal to the number of right parentheses, which is $n + m + 1$.
- (ii) The expression $E : (\neg(A \wedge B) \rightarrow C)$ is a proposition.

The proof is based upon Definition 1.2.1.

- step 1 : $A \wedge B$ is a proposition by Definition 1.2.1 (ii)
- step 2 : $\neg(A \wedge B)$ is a proposition by Definition 1.2.1 (ii)
- step 3 : C is a proposition by Definition 1.2.1 (i)
- step 4 : $(\neg(A \wedge B) \rightarrow C)$ is a proposition by Definition 1.2.1 (ii)

- (iii) The expressions below are not propositions:

- \neg : the symbol ‘ \neg ’ is not an atom
- $\wedge \rightarrow A$: the symbol ‘ \wedge ’ is not a proposition
- $(A \wedge B$: there is a deficient use of parentheses

■ 1.2.5

Example 1.2.6: The proposition F of our everyday speech is given:

F : “If it doesn’t rain, I will go for a walk”

To formalize the above proposition in the context of PL, we can use auxiliary propositional symbols:

- A : “It’s raining”
- B : “I’m going for a walk”

Then F becomes $((\neg A) \rightarrow B)$, which is a proposition.

If there is no risk of confusion whatsoever, the parentheses can be omitted:

$$F : \neg A \rightarrow B \quad \blacksquare \quad 1.2.6$$

Each atom which occurs in a well-formed formula D is considered a subformula of D . For example, A and B are subformulae of F in Example 1.2.6. Moreover, each compound well-formed part of D is also a subformula of D . For example, if

$$D : (A \wedge \neg B) \rightarrow [(C \vee \neg A) \rightarrow B]$$

then $\neg A$, $C \vee \neg A$, $A \wedge \neg B$ as well as $(C \vee \neg A) \rightarrow B$ are subformulae of D . Moreover, D itself, i.e., $(A \wedge \neg B) \rightarrow [(C \vee \neg A) \rightarrow B]$, is considered a subformula of D .

The set $\text{subform}(\sigma)$ of all subformulae of the well-formed formula σ is given by the following inductive definition:

Definition 1.2.7:

- (1) if σ is an atom A , then $\text{subform}(\sigma) = \{A\}$
- (2) if σ has the form $\neg \tau$, then $\text{subform}(\sigma) = \text{subform}(\tau) \cup \{\sigma\}$
- (3) if σ has the form $\tau \circ \varphi$, where $\circ \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$, then
 $\text{subform}(\sigma) = \text{subform}(\tau) \cup \text{subform}(\varphi) \cup \{\sigma\}$. $\blacksquare \quad 1.2.7$

Remark 1.2.8: To avoid confusion by using connectives in formulae without parentheses, we consider \neg to bind more strongly than all the other connectives, \vee and \wedge to bind more strongly than \leftrightarrow and \rightarrow , and \leftrightarrow to bind more strongly than \rightarrow . Thus

$$\neg A \rightarrow B \vee C, \quad A \wedge B \rightarrow C, \quad A \rightarrow B \leftrightarrow C$$

read respectively

$$(\neg A) \rightarrow (B \vee C), \quad (A \wedge B) \rightarrow C, \quad A \rightarrow (B \leftrightarrow C) \quad \blacksquare \quad 1.2.8$$

Remark 1.2.9: For historical reasons we will consider the symbol “ \leftarrow ” to be a logical connective, “ $B \leftarrow A$ ” meaning exactly the same as $A \rightarrow B$. $\blacksquare \quad 1.2.9$

1.3 Semantic Concepts in Propositional Logic

Valuations and truth valuations

Propositions, according to Definition 1.2.1, are general and abstract syntactic objects. We wish to interpret these abstract objects by means of semantics. This means that we are interested in determining the conditions under which a proposition is true or false. We therefore create a structure, the domain of which is $\{t, f\}$, with t, f denoting respectively the truth values “truth”, and “falsehood”, and we try to assign one of these values to each proposition. The method used in the allocation of the truth values, as well as the necessary definitions, constitute the PL semantics.

Definition 1.3.1: A **valuation** is any function:

$$F : Q \mapsto \{t, f\}$$

where Q is the set of the atoms of the language.

■ 1.3.1

A valuation thus assigns truth values to the atoms of the language.

We now provide the set $\{t, f\}$ of truth values with internal algebraic operations, so as to transform it to an algebraic structure. The internal operators of t, f which will be defined, i.e., \sim , \sqcup , \sqcap , \rightsquigarrow and $\rightsquigarrow\rightsquigarrow$, will correspond to the logical connectives \neg , \wedge , \vee , \rightarrow and \leftrightarrow . The similarity of the corresponding symbols is not fortuitous, it outlines the correspondence. Consider, for instance, the proposition $A \vee B$. By assigning certain truth values to A and B , we interpret $A \vee B$ using the interpretations of A and B . The operation which does this is \sqcup , as we will see more clearly in the next definition.

Definition 1.3.2: The internal operations \sim , \sqcup , \sqcap , \rightsquigarrow and $\rightsquigarrow\rightsquigarrow$ of $\{t, f\}$ are defined by the following tables.

	\sim
t	f
f	t

	\sqcup	t	f
t	t	t	t
f	f	t	f

	\sqcap	t	f
t	t	f	f
f	f	f	f

\leadsto	t	f
t	t	f
f	t	t

\leftrightarrow	t	f
t	t	f
f	f	t

In the tables defining \sqcap , \sqcup , \leadsto and \leftrightarrow , the first column defines the first component and the first row defines the second component of the corresponding operation. ■ 1.3.2

The structure $(\{t, f\}, \sim, \sqcup, \sqcap)$ with the operations \sim , \sqcup , \sqcap being defined by these tables, is a two-valued Boolean Algebra.

Boolean Algebras are of great importance to PL semantics [RaSi70, Curr63, Rasi74, Raut79] as well as to the general studies of the theoretical foundation of computer science.

Definition 1.3.3: Let S be the set of the propositions of our language. By **truth valuation** or **Boolean valuation** we mean any function:

$$V : S \mapsto \{t, f\}$$

such that, for every $\sigma, \tau \in S$:

- (a) if σ is an atom then $V(\sigma) \in \{t, f\}$
- (b) $V(\neg\sigma) = \sim V(\sigma)$
- (c) $V(\sigma \vee \tau) = V(\sigma) \sqcup V(\tau)$
- (d) $V(\sigma \wedge \tau) = V(\sigma) \sqcap V(\tau)$
- (e) $V(\sigma \rightarrow \tau) = V(\sigma) \leadsto V(\tau)$
- (f) $V(\sigma \leftrightarrow \tau) = V(\sigma) \leftrightarrow V(\tau)$

■ 1.3.3

As we can see in the last definition, the values of the truth valuations of the atoms of some proposition are interconnected with the algebraic operations \sim , \sqcup , \sqcap , \leadsto and \leftrightarrow , and provide the value of the truth valuation of the proposition under consideration [Smul68].

A valuation assigns a truth value, t or f , to the atoms of the language. A truth valuation is the extension of a valuation to the set of the propositions of the language. As proved in the following theorem, each truth valuation extends uniquely to a valuation on the set of the language's propositions.

Theorem 1.3.4: *For each valuation F , there is one and only one truth valuation V such that V extends F .*

Proof: We will use induction on the length of the propositions.

Let F be a valuation and Q the set of the atoms. We define inductively a truth valuation V in the following way:

$$(a) \quad V(A) = F(A) \text{ for every } A \in Q$$

Let σ, φ be two propositions. If $V(\sigma)$ and $V(\varphi)$ are already defined by (a), we impose:

$$(b) \quad V(\neg\sigma) = \sim V(\sigma)$$

$$(c) \quad V(\sigma \vee \varphi) = V(\sigma) \sqcup V(\varphi)$$

$$(d) \quad V(\sigma \wedge \varphi) = V(\sigma) \sqcap V(\varphi)$$

$$(e) \quad V(\sigma \rightarrow \varphi) = V(\sigma) \rightsquigarrow V(\varphi)$$

$$(f) \quad V(\sigma \leftrightarrow \varphi) = V(\sigma) \rightsquigarrow V(\varphi)$$

V is obviously a truth valuation extending F .

What is now left to prove is that if the truth valuations V_1 and V_2 are extensions of F , then $V_1(\varphi) = V_2(\varphi)$ for any proposition φ . The property P used for the induction is:

$$P(\varphi) : V_1(\varphi) = V_2(\varphi)$$

(a) For every propositional symbol A , we have $V_1(A) = V_2(A)$, since V_1, V_2 are extensions of F .

(b) Let us suppose that $V_1(\sigma) = V_2(\sigma)$ and $V_1(\varphi) = V_2(\varphi)$, then

$$V_1(\neg\sigma) = \sim V_1(\sigma) = \sim V_2(\sigma) = V_2(\neg\sigma)$$

$$V_1(\sigma \wedge \varphi) = V_1(\sigma) \sqcap V_1(\varphi) = V_2(\sigma) \sqcap V_2(\varphi) = V_2(\sigma \wedge \varphi)$$

$$V_1(\sigma \vee \varphi) = V_1(\sigma) \sqcup V_1(\varphi) = V_2(\sigma) \sqcup V_2(\varphi) = V_2(\sigma \vee \varphi).$$

Treating similarly the other conditions of Definition 1.3.3, we prove that V_1 and V_2 have the same value for every proposition; they therefore coincide. ■ 1.3.4

The next useful corollary is a direct consequence of Theorem 1.3.4:

Corollary 1.3.5: *Let σ be a proposition containing only the atoms A_1, \dots, A_k . If two truth valuations take the same values in the set $\{A_1, \dots, A_k\}$, i.e.,*

$$V_1(A_1) = V_2(A_1), \dots, V_1(A_k) = V_2(A_k),$$

then $V_1(\sigma) = V_2(\sigma)$. ■ 1.3.5

Example 1.3.6: Calculation of a truth valuation from a valuation:

Let S be the set of atomic propositions $S = \{A_1, A_2\}$, and F be a valuation such that:

$$F(A_1) = t$$

$$F(A_2) = f$$

By Theorem 1.3.4, the truth valuation V_F extending F which we are seeking is uniquely defined.

Let us impose:

$$V_F(A_1) := F(A_1)$$

$$V_F(A_2) := F(A_2)$$

where by “ $:=$ ” is meant “equal by definition”.

We are now able to calculate the values of the truth valuation V_F for any set of propositions which contains only the atoms A_1 and A_2 :

$$V_F(A_1 \wedge A_2) = V_F(A_1) \sqcap V_F(A_2) = t \sqcap f = f$$

$$V_F(A_1 \vee A_2) = V_F(A_1) \sqcup V_F(A_2) = t \sqcup f = t$$

$$V_F((A_1 \vee A_2) \rightarrow A_2) = V_F(A_1 \vee A_2) \rightsquigarrow V_F(A_2) = t \rightsquigarrow f = f, \text{ etc..}$$

■ 1.3.6

We can classify the PL propositions according to the truth values they are getting.

Definition 1.3.7: A proposition σ of PL is **logically true**, or a **tautology**, if *for every truth valuation* V , $V(\sigma) = t$ holds. This is denoted by $\models \sigma$. We shall write $\not\models \sigma$ to mean that σ is not a tautology, i.e., that *there exists a truth valuation* V such that $V(\sigma) = f$.

A proposition σ is **satisfiable** or **verifiable** if *there exists a truth valuation* V such that $V(\sigma) = t$.

A proposition σ such that, *for every truth valuation* V , $V(\sigma) = f$, is called **logically false**, or **not verifiable**, or is said to be a **contradiction**. ■ 1.3.7

Remark 1.3.8: Let V be a truth valuation, and

$$S_V = \{\sigma \in \text{PL propositions} \mid V(\sigma) = t\}$$

be a set consisting of the propositions of our language satisfiable by V . If for every truth valuation V , the proposition φ belongs to the corresponding set S_V , then φ is a tautology. Every S_V set, with V being a truth valuation, constitutes a possible world [Fitt69, Chel80] of the set of PL propositions. Every possible world is **Aristotelian**, meaning that for every proposition σ , only one from σ , $\neg\sigma$ can be true in a certain possible world. ($V(\sigma) = t$ or $V(\sigma) = f$, thus $V(\neg\sigma) = t$ according to Aristotle's principle of the **Excluded Middle**, Remark 1.4.4.)

Possible worlds are the basic concept of **Kripke semantics**. Kripke semantics is used in the studies of Modal Logic. Modal Logic contains special symbols -- besides the logical connectives -- called modal operators, such as \Diamond for instance, that widen the expressive capacity of the language. Thus in a modal language, besides the expression A , there is also the expression $\Diamond A$, which can be interpreted as "sometimes A is true", or " A will be valid in the future".

Modal logic allows us to express the specific properties of the statement of a program which are related to the place, the ordering, of the statement within the program. For example, consider the following FORTRAN program which calculates $n!$ for $1 \leq n \leq 10$.

```

      k = 1
      do 10 i = 1,...,10
        k = k * i
10    write (*,*)k
      end

```

Let A and B be the propositions of the modal logic

$A := \text{write } (*,*)k$ and $B := \text{end}$

When the program starts running, $\Diamond A$ and $\Diamond B$ are valid, which means that the program will find $n!$ at some point in the future and that at another future moment the program will end. ■ 1.3.8

Definition 1.3.9: Two propositions σ and τ , such that $V(\sigma) = V(\tau)$ for every truth valuation V , are said to be **logically equivalent**. This is denoted by $\sigma \equiv \tau$.

■ 1.3.9

Example 1.3.10: The propositions $A \vee \neg A$ and $((A \rightarrow B) \rightarrow A) \rightarrow A$ are tautologies.

Proof: First, we will prove that $A \vee \neg A$ is a tautology. Let V_1 and V_2 be two truth valuations such that:

$$V_1(A) = t \quad \text{and} \quad V_2(A) = f$$

We notice that:

$$\begin{aligned} V_1(A \vee \neg A) &= V_1(A) \sqcup V_1(\neg A) = V_1(A) \sqcup (\sim V_1(A)) = t \sqcup (\sim t) \\ &= t \sqcup f = t \\ V_2(A \vee \neg A) &= V_2(A) \sqcup V_2(\neg A) = V_2(A) \sqcup (\sim V_2(A)) = f \sqcup (\sim f) \\ &= f \sqcup t = t \end{aligned}$$

A random truth valuation V on A agrees, either with V_1 or with V_2 .

Thus, based on Corollary 1.3.5, we have:

$$\begin{aligned} V(A \vee \neg A) &= V_1(A \vee \neg A) = t, & \text{or} \\ V(A \vee \neg A) &= V_2(A \vee \neg A) = t \end{aligned}$$

So, the proposition is true for any truth valuation V . Then it must be a tautology.

We will now prove that $((A \rightarrow B) \rightarrow A) \rightarrow A$ is a tautology. Here we have four different valuations F_1, F_2, F_3, F_4 on $Q = \{A, B\}$

$$\begin{array}{lll} F_1(A) = t & \text{and} & F_1(B) = t \\ F_2(A) = t & \text{and} & F_2(B) = f \\ F_3(A) = f & \text{and} & F_3(B) = t \\ F_4(A) = f & \text{and} & F_4(B) = f \end{array}$$

For each of these valuations there is a unique truth valuation $V_k, k \in \{1, 2, 3, 4\}$, extending it. We can easily verify that:

$$V_k(((A \rightarrow B) \rightarrow A) \rightarrow A) = t \quad \text{for } k \in \{1, 2, 3, 4\}$$

Any truth valuation V on the propositional symbols A and B agrees with one of the $V_k, k \in \{1, 2, 3, 4\}$. By Corollary 1.3.5 we thus have:

$$V(((A \rightarrow B) \rightarrow A) \rightarrow A) = V_k(((A \rightarrow B) \rightarrow A) \rightarrow A) = t$$

for every $k \in \{1, 2, 3, 4\}$.

Hence, the proposition $((A \rightarrow B) \rightarrow A) \rightarrow A$ is true for any truth valuation and consequently it is a tautology. ■ 1.3.10

Example 1.3.11: The proposition $K \equiv [(\neg A \wedge B) \rightarrow C] \vee D$ is satisfiable.

Indeed, let F be a valuation on the propositional symbols of K , such that:

$$F(A) = t \quad F(B) = t \quad F(C) = f \quad F(D) = f$$

The truth valuation extending F takes the following values:

$$V(A) = t \quad V(B) = t \quad V(C) = f \quad V(D) = f$$

Then

$$V(\neg A) = \sim V(A) = f$$

$$V(\neg A \wedge B) = V(\neg A) \sqcap V(B) = f \sqcap t = f$$

$$V[(\neg A \wedge B) \rightarrow C] = V(\neg A \wedge B) \rightsquigarrow V(C) = f \rightsquigarrow f = t$$

$$V[((\neg A \wedge B) \rightarrow C) \vee D] = V[(\neg A \wedge B) \rightarrow C] \sqcup V(D) = t \sqcup f = t$$

meaning that $V(K) = t$.

Proposition K is not a tautology:

Let G be a valuation, with

$$G(A) = t \quad G(B) = t \quad G(C) = f \quad G(D) = f$$

Let G' be the truth valuation extending G . Using the same method used in proving that $V(K) = t$, we can prove that $G'(K) = f$. Then by Definition 1.3.7, K is not a tautology. ■ 1.3.11

1.4 Truth Tables

In the preceding section, we gave the definition of the valuation of the atoms of the language and then, by extending the valuation from atoms to compound propositions, we defined truth valuations. With this method, we wish to find out if a proposition is a tautology or a contradiction, or if it is satisfiable. But the more compound a proposition gets, the more complicated the method turns out to be.

To make things simpler, we gather all the possible valuations of a proposition's atoms in a table called the proposition's truth table. And so, for the compound propositions $\neg A$, $A \vee B$, $A \wedge B$, $A \rightarrow B$ and $A \leftrightarrow B$, we have the following truth tables:

A	$\neg A$
<i>t</i>	<i>f</i>
<i>f</i>	<i>t</i>

A	B	$A \vee B$
<i>t</i>	<i>t</i>	<i>t</i>
<i>t</i>	<i>f</i>	<i>t</i>
<i>f</i>	<i>t</i>	<i>t</i>
<i>f</i>	<i>f</i>	<i>f</i>

A	B	$A \wedge B$
<i>t</i>	<i>t</i>	<i>t</i>
<i>t</i>	<i>f</i>	<i>f</i>
<i>f</i>	<i>t</i>	<i>f</i>
<i>f</i>	<i>f</i>	<i>f</i>

A	B	$A \rightarrow B$
<i>t</i>	<i>t</i>	<i>t</i>
<i>t</i>	<i>f</i>	<i>f</i>
<i>f</i>	<i>t</i>	<i>t</i>
<i>f</i>	<i>f</i>	<i>t</i>

A	B	$A \leftrightarrow B$
<i>t</i>	<i>t</i>	<i>t</i>
<i>f</i>	<i>f</i>	<i>t</i>
<i>f</i>	<i>t</i>	<i>f</i>
<i>t</i>	<i>f</i>	<i>f</i>

Each row in these tables corresponds to a valuation and its unique extension to a truth valuation. For example, the second row of the table $A \wedge B$ is *t*, *f*, *f*, according to which *A* takes value *t*, the first element of the row, and *B* takes *f*, the second element of the row. Based on Definition 1.3.2, as well as on Definition 1.3.3, we know that the corresponding truth value of $A \wedge B$ is *t* \square *f*, namely *f*. We thus find *f*, the third element of the row.

From now on, we will use the above truth tables as definitions of the values resulting from the use of logical connectives, without referring to valuations and truth valuations.

Remark 1.4.1:

- (1) The PL disjunction is inclusive; that is $A \vee B$ can take value t also when both A and B have value t , as opposed to the everyday speech disjunction which is often exclusive: for instance, we say “I will stay home or I will go to the movies”, and by that we mean choosing one of the options and not doing both.
- (2) The proposition $A \rightarrow B$ takes value f only if A has value t and B has value f . Thus, if A is f , $A \rightarrow B$ is t , whatever the value of B is. These properties of \rightarrow and consequently the part of Propositional Logic which is based on these same properties, have not always been accepted by all the logic schools.

Back in 1910-1920, the prevailing concept about applied sciences in Europe was that of the Hilbert school (Hilbert 1862-1953) [Boye68, Heij67]:

“Mathematics and Logic must be axiomatically founded, and the working methods within the limits of these sciences have to be extremely formal.”

But in 1918, Brouwer (1881-1966) published a very severe criticism of PL. The alternative system recommended in his work was called **intuitionistic logic**. This name finds its origin in the fact that Brouwer stated, based on Kant’s belief, that we perceive the natural numbers (and consequently the logic which characterises the sciences) only by means of our intuition [Dumm77, Brou75]. Intuitionistic logic never fully replaced the logic founded in Hilbert’s concepts (see Remark 1.8.8), however, nowadays it has several applications in Computer Science.

On the other hand, PL offers a very effective instrument (in the Aristotelian sense) for studying and solving problems, and constitutes the foundation of today’s technology and of the sciences generally. ■ 1.4.1

Let us recapitulate what we have seen about the truth tables of a proposition:

Based on the inductive definition of propositions as well as on the definition of the values of the logical connectives, we can construct the truth table of any proposition by assigning values to the atoms used in its formation.

Example 1.4.2: The truth table of the proposition $A \wedge B \rightarrow C$ is:

A	B	C	$A \wedge B$	$A \wedge B \rightarrow C$
t	t	t	t	t
t	t	f	t	f
t	f	t	f	t
t	f	f	f	t
f	t	t	f	t
f	t	f	f	t
f	f	t	f	t
f	f	f	f	t

For the triplet of atoms (A, B, C) , the triplet of truth values (f, t, f) makes $A \wedge B \rightarrow C$ true, whereas the triplet (t, t, f) makes it false. ■ 1.4.2

The **short truth table** of the proposition $A \wedge B \rightarrow C$ is the truth table remaining when the fourth auxiliary column is omitted.

The short truth table of a proposition containing n atoms consists of 2^n rows and $n + 1$ columns.

By means of the truth tables, we can determine whether a proposition is true or false. By Corollary 1.3.5, if the truth tables of two propositions coincide in the columns of their atom's values and in their last columns, then the two propositions are logically equivalent.

Example 1.4.3:

(i) The proposition $((A \rightarrow B) \rightarrow A) \rightarrow A$ is a tautology

A	B	$A \rightarrow B$	$((A \rightarrow B) \rightarrow A)$	$((A \rightarrow B) \rightarrow A) \rightarrow A$
t	t	t	t	t
t	f	f	t	t
f	t	t	f	t
f	f	t	f	t

(ii) The proposition $(P \rightarrow Q) \wedge (P \wedge \neg Q)$ is not verifiable

P	Q	$\neg Q$	$P \rightarrow Q$	$P \wedge \neg Q$	$(P \rightarrow Q) \wedge (P \wedge \neg Q)$
t	t	f	t	f	f
t	f	t	f	t	f
f	t	f	t	f	f
f	f	t	t	f	f

■ 1.4.3

Remark 1.4.4: Based on Definition 1.3.7, we have:

- (i) A proposition is a tautology if and only if its negation is not satisfiable.
- (ii) A proposition is satisfiable if and only if its negation is not a tautology.
- (iii) A proposition which is a tautology is satisfiable, whereas a satisfiable proposition is not necessarily a tautology.
- (iv) There are certain basic tautologies which are often used:

- | | |
|---|-------------------------|
| (1) $\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$ | De Morgan Law |
| (2) $\neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$ | De Morgan Law |
| (3) $\neg(\neg A) \leftrightarrow A$ | Double Negation Law |
| (4) $(A \rightarrow B) \leftrightarrow (\neg B \rightarrow \neg A)$ | Contrapositive Law |
| (5) $(B \rightarrow C) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$ | First Syllogistic Law |
| (6) $(A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C))$ | Second Syllogistic Law |
| (7) $(A \rightarrow (B \rightarrow C)) \leftrightarrow ((A \wedge B) \rightarrow C)$ | Transportation Law |
| (8) $A \vee \neg A$ | The Excluded Middle Law |

Aristotle was the first to present propositions (5), (6) and (8). Propositions (5) and (6) are even referred to as Aristotle's Syllogistic Laws. ■ 1.4.4

The method of finding the value of a compound proposition with the use of truth tables is quite simple, as long as the proposition contains a small number of atoms. In the case of a proposition with three atoms, the corresponding truth table has $2^3 = 8$ rows. The truth table of a proposition with 4 atoms already has $2^4 = 16$ rows, and with 10 atoms, it has $2^{10} = 1024$ rows!

Furthermore, we cannot use truth tables in Predicate Logic. We will thus study, in the following sections, more advanced and economical methods of determining the truth value of a proposition or a set of propositions. These methods will constitute the base for the study of Logic Programming.

1.5 Consequences and Interpretations

In Example 1.4.2 of the truth table of proposition $(A \wedge B) \rightarrow C$ we saw that this proposition takes value t when all three atoms A, B and C take value t . This is to say that the validity of $(A \wedge B) \rightarrow C$ was a consequence of the fact that every proposition of $S = \{A, B, C\}$ took value t [Chur56]. We thus give the following definition:

Definition 1.5.1: Let S be a set of propositions. A proposition σ is called a **consequence of S** (denoted by $S \models \sigma$), if for every truth valuation V , for which $V(\varphi) = t$ for any $\varphi \in S$, we can deduce that $V(\sigma) = t$ holds.

The set $Con(S) = \{\sigma \mid S \models \sigma\}$ is the set of all consequences of S . Formally:

$$\begin{aligned} \sigma \in Con(S) \quad \Leftrightarrow \quad & \text{(for every truth valuation } V) \\ & \text{(for every } \varphi \in S) \\ & (V(\varphi) = t \Rightarrow V(\sigma) = t) \end{aligned}$$

Instead of (for every $\varphi \in S$) $(V(\varphi) = t)$ we often write $V[S] = \{t\}$ or even informally $V[S] = t$. ■ 1.5.1

Remark 1.5.2: The symbols “ \Leftrightarrow ” and “ \Rightarrow ” used in the above definition as “if and only if” and “implies” are symbols of the **metalanguage**. The metalanguage is the language used to reason about PL formulae and to investigate their properties. Therefore, when we for instance say $\models \varphi$, proposition φ is a tautology, we express a judgement about φ . “ $\models \varphi$ ” is a **metaproposition** of PL, namely a proposition of the PL metalanguage.

The metalanguage can also be formalized, just like the PL language. In order to avoid excessive and pedantic use of symbols, like for example \rightarrow of the language and \Rightarrow of the metalanguage, we use as metalanguage, a carefully and precisely formulated form of the spoken language.

We will end our comment about the metalanguage with another example: Let A be a proposition of PL. Then the expression “ $A \rightarrow A$ ” is a PL proposition, whereas “if A is a PL proposition, then A is a PL proposition” is a proposition of the PL metalanguage. ■ 1.5.2

Example 1.5.3: Let $S = \{A \wedge B, B \rightarrow C\}$ be a set of propositions. Then proposition C is a consequence of S , i.e., $S \models C$.

Proof: Let us suppose that V is a truth valuation validating all S propositions:

$$V(A \wedge B) = t \quad \text{and} \quad V(B \rightarrow C) = t.$$

Then we have, by Definition 1.3.3:

$$V(A) \sqcap V(B) = t \quad (1) \quad \text{and} \quad V(B) \rightsquigarrow \vee(C) = t \quad (2)$$

Then by (1) and Definition 1.3.2 we have

$$V(A) = V(B) = t \quad (3)$$

(2) thus becomes

$$t \rightsquigarrow V(C) = t \quad (4)$$

By Definition 1.3.2 and (4) we can conclude $V(C) = t$. That means that every truth valuation verifying all S propositions, verifies C too. C is therefore a consequence of S , $S \models C$. ■ 1.5.3

If we denote by *Taut* the set of all tautologies, we can prove the following corollary.

Corollary 1.5.4: $Con(\emptyset) = Taut$, where \emptyset is the empty set.

Proof: Consider $\sigma \in Taut$. Then for every truth valuation V , $V(\sigma) = t$. Then for every V such that $\underline{V[\emptyset] = \{t\}}$,

$$\underline{V[\emptyset] = \{t\}} \Rightarrow V(\sigma) = t$$

Then $\sigma \in Con(\emptyset)$.

Conversely: Consider $\sigma \in Con(\emptyset)$. Then for every truth valuation V we have:

$$\text{For every } \underline{\varphi \in \emptyset}, \text{ if } V(\varphi) = t \text{ then } V(\sigma) = t \quad (1)$$

But \emptyset has no elements. We can thus write:

$$\text{For every } \underline{\varphi \in \emptyset}, \text{ if } V(\varphi) = f \text{ then } V(\sigma) = t \quad (2)$$

By (1) and (2) we have $V(\sigma) = t$, whatever the value of $\varphi \in \emptyset$. This means that for all truth valuations V , $V(\sigma) = t$ holds. σ is thus a tautology. ■ 1.5.4

Thus tautologies are not consequences of any particular set S , they are independent from S , for every set S of propositions, and they are only related to the truth tables of section 1.3.

The underlined parts in the Proof of Corollary 1.5.4 reveal three very technical parts of the proof. The empty set has no elements. How can we therefore write the implications with the underlinings? Let us think of the third and fourth line of the definition of the truth table of proposition $A \rightarrow B$ (see page 17). If A has value f and B value t or value f , then $A \rightarrow B$ is t . This means that an implication $A \rightarrow B$, with A being false, is always true! This is exactly what we use in the metalanguage, the language in which our theorems are proved. In (1) for example, “ $\varphi \in \emptyset$ ” cannot be true because \emptyset has no elements. Not depending on whether “ $V(\varphi) = t$ then $V(\sigma) = t$ ” is true or false, (1) is thus true!

As mentioned before, PL deals with the study of sets of propositions. The following two definitions are related to the satisfiability of a set of propositions.

Definition 1.5.5: A set of propositions S is (semantically) **consistent**, if there is a truth valuation to verify every proposition in S . Formally:

$$\text{consistent}(S) \Leftrightarrow (\text{there is a valuation } V) [(\text{for every } \sigma \in S) (V(\sigma) = t)]$$

The fact that S is consistent is denoted by $V(S) = t$.

For the consistent set S , we also use the terms **verifiable** or **satisfiable** which have the same import.

S is correspondingly **inconsistent**, **non-verifiable** or **non-satisfiable** if for every truth valuation there is at least one non-satisfiable proposition in S .

$$\text{inconsistent}(S) \Rightarrow (\text{for every } V) [(\text{there is } \sigma \in S) (V(\sigma) = f)]$$

■ 1.5.5

Example 1.5.6: The set of propositions $S = \{A \wedge \neg B, A \rightarrow B\}$ is inconsistent.

Proof:

Let us assume there is a truth valuation V such that:

$$\text{for every } C \in S, V(C) = t$$

Then:

$$V(A \wedge \neg B) = t \quad \text{and} \quad V(A \rightarrow B) = t$$

which means that:

$$V(A) \sqcap V(\neg B) = t \quad (1) \quad \text{and} \quad V(A) \rightsquigarrow V(B) = t \quad (2)$$

By (1) and Definition 1.3.2, we have $V(A) = V(\neg B) = f$, so:

$$V(A) = t \quad \text{and} \quad V(B) = f$$

Then (2) becomes:

$$t \rightsquigarrow f = t$$

which contradicts Definition 1.3.2, where $t \rightsquigarrow f = f$. Consequently, no truth valuation can verify all propositions in S , and so S is inconsistent. ■ 1.5.6

Definition 1.5.7: A truth valuation which satisfies a set of propositions S is called an **interpretation** of S . The set of all interpretations of S is denoted by $Int(S)$, where:

$$Int(S) = \{V \mid V \text{ truth valuation and for every } \sigma \in S, V(\sigma) = t\}$$

■ 1.5.7

We now present a corollary with some useful conclusions about consequences and interpretations.

Corollary 1.5.8: For the sets of propositions S, S_1, S_2 we have:

- (1) $S_1 \subseteq S_2 \Rightarrow Con(S_1) \subseteq Con(S_2)$
- (2) $S \subseteq Con(S)$
- (3) $Taut \subseteq Con(S)$, for any set of propositions S
- (4) $Con(S) = Con(Con(S))$

- (5) $S_1 \subseteq S_2 \Rightarrow Int(S_2) \subseteq Int(S_1)$
- (6) $Con(S) = \{\sigma \mid V(\sigma) = t, \text{ for every } V \in Int(S)\}$
- (7) $\sigma \in Con(\{\sigma, \dots, \sigma_n\}) \Leftrightarrow \sigma_1 \rightarrow (\sigma_2 \rightarrow (\sigma_n \rightarrow \sigma) \dots) \in Taut$

Proof:

- (1) Let us assume $\sigma \in Con(S_1)$. Let V be a truth valuation such that for every $\varphi \in S_2$, $V(\varphi) = t$ holds. We thus have $V(\varphi) = t$ for every $\varphi \in S_1$, (since $S_1 \subseteq S_2$). Then $V(\sigma) = t$, since $\sigma \in Con(S_1)$. Hence $Con(S_1) \subseteq Con(S_2)$.
- (2) If $\sigma \in S$, then every truth valuation V which validates all the propositions of S also validates σ . Thus $\sigma \in Con(S)$. But then $S \subseteq Con(S)$.
- (3) Let us assume that $\sigma \in Taut$. Let V be a truth valuation such that, for every $\varphi \in S$, $V(\varphi) = t$ holds. Then $V(\sigma) = t$, thus $\sigma \in Con(S)$. Hence $Taut \subseteq Con(S)$.
- (4) By (2), we have $S \subseteq Con(S)$. By (1), $Con(S) \subseteq Con(Con(S))$. We just need to prove

$$Con(Con(S)) \subseteq Con(S)$$

Let us assume that $\sigma \in Con(Con(S))$. Let V be a truth valuation such that for every $\varphi \in S$, $V(\varphi) = t$. Then for every $\tau \in Con(S)$, $V(\tau) = t$. By the definition of $Con(Con(S))$, we thus have $V(\sigma) = t$ which means that $\sigma \in Con(S)$. Hence $Con(Con(S)) \subseteq Con(S)$.

- (5) If $V \in Int(S_2)$, then for every $\sigma \in S_2$ we have $V(\sigma) = t$. Since $S_1 \subseteq S_2$, for every $\sigma \in S_1$, $V(\sigma) = t$ holds, and hence $V \in Int(S_1)$.
- (6) If $\sigma \in Con(S)$, then for every $V \in Int(S)$, $V(\sigma) = t$. Then

$$\sigma \in \{\varphi \mid V(\varphi) = t, \text{ for every } V \in Int(S)\}$$

Furthermore, if

$$\varphi \in \{\sigma \mid V(\sigma) = t, \text{ for every } V \in Int(S)\}$$

then it is obvious that $\varphi \in Con(S)$.

(7) (\Rightarrow) Let us assume that $\varphi \in \text{Con}(\{\sigma_1, \sigma_2, \dots, \sigma_n\})$. Let V be a truth valuation. For V , we have the following possibilities:

- (a) For each σ_i , $1 \leq i \leq n$, $V(\sigma_i) = t$
- (b) There is at least one σ_j , $1 \leq j \leq n$, such that $V(\sigma_j) = f$.

We shall analyse these two cases separately.

- (a) Since for every σ_i , with $1 \leq i \leq n$,

$$V(\sigma_i) = t \quad \text{and} \quad \varphi \in \text{Con}(\{\sigma_1, \dots, \sigma_n\})$$

we have $V(\varphi) = t$. Thus

$$V(\sigma_n \rightarrow \varphi) = V(\sigma_n) \rightsquigarrow V(\varphi) = t \rightsquigarrow t = t$$

If $V(\sigma_k \rightarrow (\sigma_{k+1} \rightarrow (\sigma_n \rightarrow \varphi) \dots)) = t$, then

$$\begin{aligned} & V(\sigma_{k-1} \rightarrow (\sigma_k \rightarrow (\sigma_{k+1} \rightarrow \dots \rightarrow (\sigma_n \rightarrow \varphi) \dots))) \\ &= V(\sigma_{k-1}) \rightsquigarrow V(\sigma_k \rightarrow (\sigma_{k+1} \rightarrow \dots \rightarrow (\sigma_n \rightarrow \varphi) \dots)) \\ &= t \rightsquigarrow t = t \end{aligned}$$

Hence (inductively), $V(\sigma_1 \rightarrow (\sigma_2 \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots)) = t$.

- (b) Let r be the least natural number for which $V(\sigma_r) = f$. Thus

$$\begin{aligned} & V(\sigma_r \rightarrow (\sigma_{r+1} \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots)) \\ &= V(\sigma_r) \rightsquigarrow V(\sigma_{r+1} \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots) \\ &= f \rightsquigarrow V(\sigma_{r+1} \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots) = t \end{aligned}$$

Since $V(\sigma_{r-1}) = t$, we have

$$\begin{aligned} & V(\sigma_{r-1} \rightarrow (\sigma_r \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots)) \\ &= V(\sigma_{r-1}) \rightsquigarrow V(\sigma_r \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots) = t \rightsquigarrow t = t \end{aligned}$$

If $V(\sigma_k \rightarrow (\sigma_{k+1} \rightarrow \dots (\sigma_r \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots) \dots)) = t$, then

$$\begin{aligned} & V(\sigma_{k-1} \rightarrow (\sigma_k \rightarrow (\sigma_{k+1} \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots))) \\ &= V(\sigma_{k-1}) \rightsquigarrow V(\sigma_k \rightarrow (\sigma_{k+1} \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots)) \\ &= t \rightsquigarrow t = t \end{aligned}$$

Hence $V(\sigma_1 \rightarrow (\sigma_2 \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots)) = t$.

In (a) as well as in (b), we have: $V(\sigma_1 \rightarrow (\sigma_2 \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots)) = t$ for any truth valuation V , and hence $(\sigma_1 \rightarrow (\sigma_2 \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots))$ is a tautology.

(\Leftarrow) Let us assume that $\sigma_1 \rightarrow (\sigma_2 \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots)$ is a tautology. Let V be a truth valuation such that $V(\sigma_i) = t$ for every $i \in \{1, 2, \dots, n\}$. If we assume that $V(\varphi) = f$, then

$$V(\sigma_n \rightarrow \varphi) = V(\sigma_n) \rightsquigarrow V(\varphi) = t \rightsquigarrow f = f.$$

Therefore, if $V(\sigma_k \rightarrow (\sigma_{k+1} \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots)) = f$, then

$$\begin{aligned} V(\sigma_{k-1} \rightarrow (\sigma_k \rightarrow (\sigma_{k+1} \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots))) \\ &= V(\sigma_{k-1}) \rightsquigarrow V(\sigma_k \rightarrow (\sigma_{k+1} \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots)) \\ &= t \rightsquigarrow f = t. \end{aligned}$$

Hence $V(\sigma_1 \rightarrow (\sigma_n \rightarrow \varphi) \dots) = f$ is a contradiction because of the assumption that $(\sigma_1 \rightarrow (\sigma_2 \rightarrow \dots (\sigma_n \rightarrow \varphi) \dots))$ is a tautology.

■ 1.5.8

Corollary 1.5.8(7) provides us with a method to determine whether φ is a consequence of a finite set of propositions S , by checking in 2^n steps if the right side of (7) is a tautology. However, for an infinite set S , this method would require an infinite number of steps. The use of semantic tableaux becomes, in that case, more appropriate.

1.6 Adequacy of Logical Connectives – Normal Forms

The finding of the truth value of a proposition, as the proof of consistency or lack of consistency of a set of propositions, often depends on the number and the kind of connectives appearing in the propositions. The five logical connectives which we use are the connectives one deals with more frequently in mathematical texts. In the following paragraphs, it will be proved that any set of logical connectives can be expressed by means of the set $\{\neg, \wedge, \vee\}$, and we will thus have proven that the set of logical connectives $\{\neg, \wedge, \vee\}$ suffices to express any PL propositions [Smul68, Mend64, Schm60].

Definition 1.6.1: A set Q of logical connectives is said to be **adequate**, if for every PL proposition there is a logically equivalent proposition which does not contain connectives different from those contained in Q . ■ 1.6.1

Theorem 1.6.2: $\{\neg, \wedge, \vee\}$ is an adequate set.

Proof: Let $\sigma(A_1, A_2, \dots, A_k)$, be a proposition in which only the atoms A_1, \dots, A_k appear. Let us construct the short truth table of σ :

A_1	\dots	A_λ	\dots	A_k	$\sigma(A_1, \dots, A_k)$
σ_{1_1}	\dots	σ_{1_λ}	\dots	σ_{1_k}	σ_1
\vdots		\vdots		\vdots	\vdots
σ_{ν_1}	\dots	σ_{ν_λ}	\dots	σ_{ν_k}	σ_ν
\vdots		\vdots		\vdots	\vdots
$\sigma_{2_1^k}$	\dots	$\sigma_{2_\lambda^k}$	\dots	$\sigma_{2_k^k}$	σ_{2^ν}

σ_{n_ℓ} denotes the corresponding truth value on row n and in column ℓ , and σ_n is the truth value of $\sigma(A_1, \dots, A_k)$ in the n^{th} row.

- (1) Let us suppose that in the last column there is at least one t . We will prove that there is a proposition, the short truth table of which has the same last column as the above table, and which only uses the connectives \neg , \wedge and \vee . In that case, the proposition will be logically equivalent to $\sigma(A_1, \dots, A_k)$.

For any atom A , A^t denotes A and A^f denotes $\neg A$. From the n^{th} row we form the conjunction:

$$t_n : (A_1^{\sigma_{\nu_1}} \wedge \dots \wedge A_k^{\sigma_{\nu_k}})$$

which only contains the connectives \neg and \wedge . Let ℓ_1, \dots, ℓ_m be the rows with a t on the last column. Then the proposition:

$$t_{\ell_1} \vee \dots \vee t_{\ell_m}$$

is the proposition we are seeking since its short truth table has a t in its last column, exactly on those specific rows where the short truth table of $\sigma(A_1, \dots, A_k)$ has a t .

- (2) If the last column does not contain any t , then the proposition is false, and thus logically equivalent to $(A \wedge \neg A)$, where A is any propositional symbol. ■ 1.6.2

On that account, using the technique of the above theorem, we can express any proposition by means of the connectives \neg , \wedge and \vee .

The resulting equivalent proposition is said to be in a **Disjunctive Normal Form (DNF)**. Let F be a proposition such that the atoms of F are A_1, \dots, A_n . The general form of F in a DNF is:

$$\text{DNF}(F) : (A_{1_1} \wedge \dots \wedge A_{1_n}) \vee (A_{2_1} \wedge \dots \wedge A_{2_n}) \vee \dots \vee (A_{k_1} \wedge \dots \wedge A_{k_n})$$

where $A_{i_j} \in \{A_1, \dots, A_n\}$ or $A_{i_j} \in \{\neg A_1, \dots, \neg A_n\}$, i.e., A_{i_j} are atoms or negations of the atoms of F and in every conjunctive component of $\text{DNF}(F)$, each atom of F occurs only once, negated or unnegated.

The dual concept to DNF is called a **Conjunctive Normal Form (CNF)**, and has the following form:

$$\text{CNF}(F) : (A_{1_1} \vee \dots \vee A_{1_n}) \wedge (A_{2_1} \vee \dots \vee A_{2_n}) \wedge \dots \wedge (A_{k_1} \vee \dots \vee A_{k_n})$$

Example 1.6.3: Find the DNF of proposition F , given its short truth table:

	A	B	C	F
1	t	t	t	t
2	t	t	f	f
3	t	f	t	f
4	t	f	f	f
5	f	t	t	t
6	f	t	f	f
7	f	f	t	f
8	f	f	f	t

We follow the method described in Theorem 1.6.2.

Step 1: We find all the rows with a t in the last column. Those are rows 1, 5, 8.

Step 2: $\text{DNF}(F) \equiv t_1 \vee t_5 \vee t_8 \equiv (A \wedge B \wedge C) \vee (\neg A \wedge B \wedge C) \vee (\neg A \wedge \neg B \wedge \neg C)$

■ 1.6.3

We will now give two interesting and useful corollaries concerning the adequacy of concrete sets of logical connectives.

Corollary 1.6.4: $K_1 = \{\neg, \vee\}$ and $K_2 = \{\neg, \wedge\}$ are adequate sets.

Proof: Using truth tables, we can prove that:

$$A \rightarrow B \equiv \neg A \wedge B, \quad A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A), \quad \text{and}$$

$$A \wedge B \equiv \neg(\neg A \vee \neg B)$$

Consequently, any PL proposition can be expressed with connectives from $\{\neg, \vee\}$. K_1 is thus adequate.

We can also prove that K_2 is adequate by:

$$A \vee B \equiv \neg(\neg A \wedge \neg B), \quad A \rightarrow B \equiv \neg A \vee B, \quad \text{and}$$

$$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A) \quad \blacksquare \quad 1.6.4$$

Example 1.6.5: Apart from connectives $\neg, \wedge, \vee, \rightarrow$ and \leftrightarrow , we can create other connectives such as $|$ and $:$ defined by the following truth tables:

A	B	$A B$
t	t	f
t	f	t
f	t	t
f	f	t

A	B	$A : B$
t	t	f
t	f	f
f	t	f
f	f	t

■ 1.6.5

Corollary 1.6.6: The sets $\Sigma_1 = \{ | \}$ and $\Sigma_2 = \{ : \}$ are adequate.

Proof: (Σ_1): Intuitively $A | B$ means that A and B cannot be both true. Then $A | B \equiv \neg(A \wedge B)$. (This equivalence can easily be verified with truth tables). Then:

$$A | A \equiv \neg(A \wedge A) \equiv \neg A$$

thus negation is expressed by $|$. In the same way, for conjunction

$$A \wedge B \equiv \neg\neg(A \wedge B) \equiv \neg(A | B) \equiv (A | B) | (A | B)$$

(Σ_2): The negation then becomes: $\neg A \equiv \neg(A \vee A) \equiv A : A$, and the conjunction: $A \wedge B \equiv \neg(\neg A \vee \neg B) \equiv (A : A) : (B : B)$ ■ 1.6.6

Note that $\{ | \}$ and $\{ : \}$ are the only singletons of logical connectives which are adequate (see Exercise 20). Furthermore, every set of logical connectives with two elements, one of them being \neg and the other being one of \wedge , \vee and \rightarrow is adequate, [Schm60, Smul68, Mend64].

Remark 1.6.7: The conversion of a proposition σ to another proposition φ , which is logically equivalent but containing connectives different from those in σ , is very useful. For example, in the Resolution Proof Method, which will be developed in the following sections, we express all implications, i.e., propositions of the kind $\sigma \rightarrow \varphi$, using the connectives \neg , \wedge and \vee . For this conversion we use the equivalence

$$A \rightarrow B \equiv \neg A \vee B$$

which can be verified by truth tables. ■ 1.6.7

In the following sections we will describe the proof methods used in PL. Their presentation will simplify the introduction to Logic Programming.

1.7 Semantic Tableaux

Proof methods are algorithmic procedures which we can follow to find whether a proposition is or is not a tautology and whether a set of propositions can or cannot be satisfied. These methods are included in the development of Automatic Theorem Proving, a theory which constitutes a basic application of Logic Programming.

The first method of algorithmic proofs which will be described uses semantic tableaux. Gentzen (German logician, 1909-1945) was the first to prove in 1934 that all tautologies are produced by applications of certain rules, that for every tautology φ there is a certain procedure resulting in φ , [Klee52, Raut79]. Proof theory was used in 1955 by Beth and Hintikka to create an algorithm determining whether a proposition is a tautology or not.

With the help of Beth semantic tableaux, or simply semantic tableaux, we can examine what the possibilities are that a given proposition takes truth value t or truth value f .

The semantic tableau of a compound proposition K is constructed inductively, based on the semantic tableaux of the propositions appearing in K . We thus start by defining atomic semantic tableaux [Smul68, Fitt90]:

Definition 1.7.1:

- (1) Let σ be a proposition. $f\sigma$ denotes the assertion “ σ is false” and $t\sigma$ denotes the assertion that “ σ is true”. $t\sigma$ and $f\sigma$ are called **signed formulae**.
- (2) According to the inductive definition of propositions, the atomic semantic tableaux for propositions A, σ_1, σ_2 and for the propositions formed by A, σ_1 and σ_2 are as in the following table.

1a tA	1b fA	2a $t(\sigma_1 \wedge \sigma_2)$ $t\sigma_1$ $t\sigma_2$	2b $f(\sigma_1 \wedge \sigma_2)$ / \ $f\sigma_1$ $f\sigma_2$
3a $t(\neg\sigma)$ $f\sigma$	3b $f(\neg\sigma)$ $t\sigma$	4a $t(\sigma_1 \vee \sigma_2)$ / \ $t\sigma_1$ $t\sigma_2$	4b $f(\sigma_1 \vee \sigma_2)$ $f\sigma_1$ $f\sigma_2$
5a $t(\sigma_1 \rightarrow \sigma_2)$ / \ $f\sigma_1$ $t\sigma_2$	5b $f(\sigma_1 \rightarrow \sigma_2)$ $t\sigma_1$ $f\sigma_2$	6a $t(\sigma_1 \leftrightarrow \sigma_2)$ / \ $t\sigma_1$ $f\sigma_1$ $t\sigma_2$ $f\sigma_2$	6b $f(\sigma_1 \leftrightarrow \sigma_2)$ / \ $t\sigma_1$ $f\sigma_1$ $f\sigma_2$ $t\sigma_2$

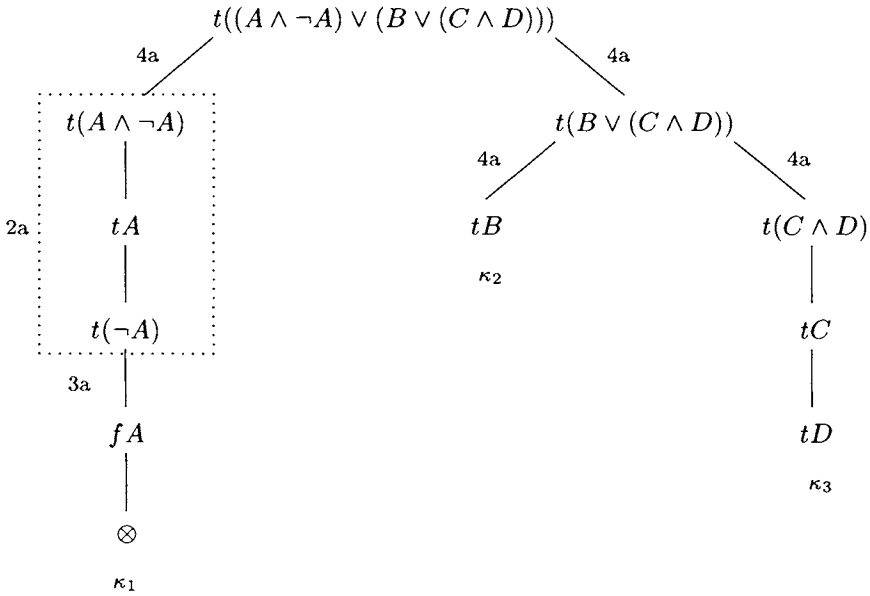
Intuitively, the signed formula tA or fA is regarded as the assertion that A is respectively true or false. In the atomic semantic tableau (4a), the assertion that $\sigma_1 \vee \sigma_2$ is true thus requires σ_1 to be true or σ_2 to be true (branching), whereas in the tableaux (5b), the assertion that $\sigma_1 \rightarrow \sigma_2$ is false requires

σ_1 to be true and σ_2 to be false (sequence). That means that in an atomic semantic tableaux, a branching denotes a disjunction whereas a sequence stands for a conjunction. ■ 1.7.1

To build the semantic tableau of a compound proposition K , we start by writing the signed formula tK or fK , at the origin of the semantic tableau. We then unfold the semantic tableau of K , according to Definition 1.7.1.

We now give an example which will be followed by the formal definition of the general rule for construction of semantic tableaux.

Example 1.7.2: Let $K : (A \wedge \neg A) \vee (B \vee (C \wedge D))$ be a proposition. The atoms of K are A, B, C and D . We start the semantic tableau with origin tK .



This is a complete semantic tableau with three branches, namely three sequences κ_1 , κ_2 and κ_3 . The branches start from the origin. The left branch, κ_1 , is contradictory since it contains the mutually contradictory signed formulae tA and fA . The contradiction of a branch is denoted by the symbol \otimes at the bottom of the branch. The other two branches are not contradictory.

By means of the semantic tableau of K , we see that the tK hypothesis is correct under certain conditions, such as for instance tB in the branch κ_2 or tD in the branch κ_3 . It is however sometimes false, as in branch κ_1 .

It is easy to set apart the constituent atomic semantic tableaux from the above semantic tableau. For example:

$$\begin{array}{c}
 t(A \wedge \neg A) \\
 | \\
 tA \\
 | \\
 t(\neg A)
 \end{array}$$

in the dotted area is the semantic tableau 2a.

■ 1.7.2

We now define the concepts needed for the construction of semantic tableaux.

Definition 1.7.3:

- (1) The **nodes** of a semantic tableau are all the signed formulae which occur in the table.
- (2) A node of a semantic tableau is said to be **used** if it occurs as the origin of an atomic semantic tableau. Otherwise it is said to be **unused**.
- (3) A branch of a semantic tableau is said to be **contradictory** if for a certain proposition σ , $t\sigma$ and $f\sigma$ are nodes of the branch.
- (4) A semantic tableau is said to be **complete** if none of the non-contradictory branches has unused nodes. Otherwise it is said to be **incomplete**.
- (5) A semantic tableau is **contradictory** if all its branches are contradictory.

■ 1.7.3

Definition 1.7.4: Inductive construction of semantic tableaux:

We will construct a **semantic tableau** for proposition K as follows:

We start with the signed formula tK (or fK) as the origin and we proceed inductively.

Step n : We have an atomic semantic tableau T_n .

Step $n + 1$: The atomic semantic tableau T_n will be extended to tableau T_{n+1} by using certain nodes of T_n which will not be used again. From the unused T_n

nodes nearest to the origin, we select the one most on the left. Let X be that node.

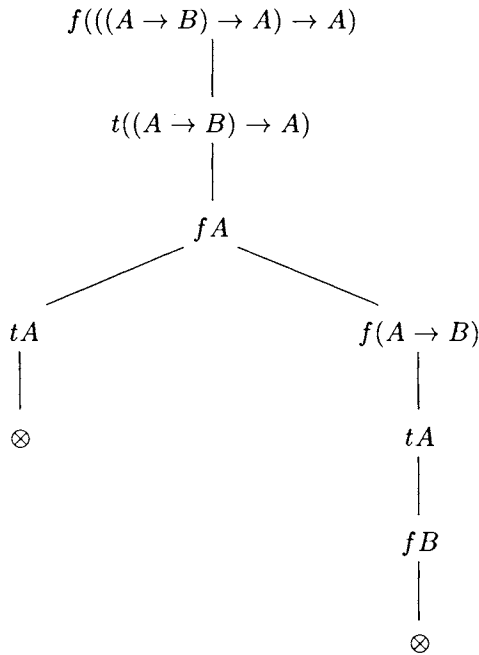
We now extend every non-contradictory branch passing through X by concatenating an atomic semantic tableau with an X origin at the end of each of these branches. What comes as a result is the semantic tableau T_{n+1} (in practice, one avoids rewriting the node X , since it is already a node of the non-contradictory branch).

The construction finishes when every non-contradictory branch has no unused nodes. ■ 1.7.4

The following example, a contradictory semantic tableau, clarifies the above construction.

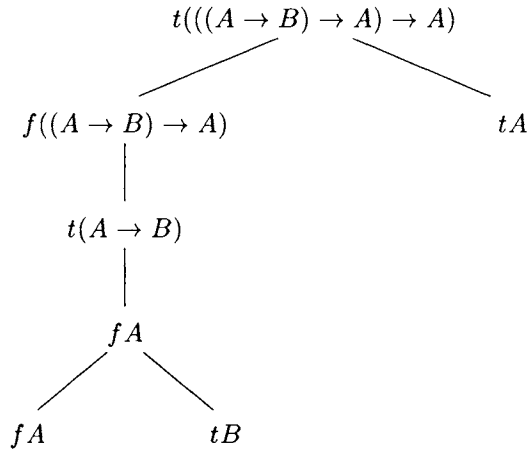
Example 1.7.5: Peirce's Law $((A \rightarrow B) \rightarrow A) \rightarrow A$

(1) We start with the assertion that $((A \rightarrow B) \rightarrow A) \rightarrow A$ is false:



The claim that Peirce's Law is false has led us to a contradictory semantic tableau, therefore the corresponding formula is true.

- (2) If we start with the assertion that the proposition is true, the conclusion still remains the same:



We observe that there is no contradictory branch. Then, if fA or tB and fA , or tA , $((A \rightarrow B) \rightarrow A) \rightarrow A$ becomes true. Even if fB , then tA or fA holds. Hence Peirce's Law is logically true. ■ 1.7.5

Intuitively:

If a complete semantic tableau with an fK origin is found to be contradictory, that means that we have tried all possible ways to make proposition K false and we have failed. Then K is a tautology.

This central idea is expressed by the following definition:

Definition 1.7.6: A **Beth-proof** of a proposition K is a complete contradictory semantic tableau with an fK origin. A complete contradictory tableau with a tK origin is called a **Beth-refutation** of K .

Proposition K is said to be **Beth-provable** if it has a **Beth-proof**. K is called **Beth-refutable** if there is a **Beth-refutation** of K . The fact that K is Beth-provable is denoted by $\vdash_B K$. ■ 1.7.6

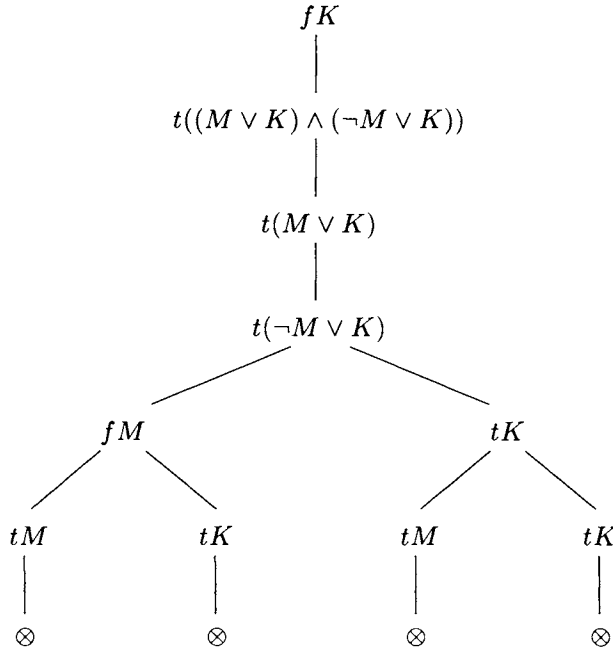
As we will prove in Theorem 1.10.7 and Theorem 1.10.9, every tautology is Beth-provable (completeness of Beth-proofs) and conversely every Beth-provable proposition is a tautology (soundness of Beth-proofs).

Example 1.7.7: Let us assume that the following propositions are true:

- (1) George loves Maria or George loves Catherine.
- (2) If George loves Maria, then he loves Catherine.

Who does George finally love?

Let us denote “George loves Maria” by M and “George loves Catherine” by K . Then, (1) $\equiv M \vee K$ and (2) $\equiv M \rightarrow K \equiv \neg M \vee K$. We form the proposition $A \equiv (M \vee K) \wedge (\neg M \vee K) \equiv (1) \wedge (2)$ which is by hypothesis true. We wish to know if George loves Catherine, or equivalently, whether tK . Let us suppose that he does not love her, that fK . We then construct a semantic tableau with an fK origin.



In step 2 we added $t((M \vee K) \wedge (\neg M \vee K))$, since (1) and (2) are given to be true. Starting with tK , we finish with a contradictory tableau, which means that proposition K is always true, in other words George loves Catherine!!!

If we construct a semantic tableau with an fM origin, we will end up with a non-contradictory tableau, and thus we will not be able to conclude whether George loves Maria or not!!

■ 1.7.7

1.8 Axiomatic Proofs

Propositional Logic, just like other mathematical systems, can also be presented as an axiomatic system with logical axioms and derivation rules instead of the semantic tableaux. The axioms are some of the tautologies, and a derivation rule R **derives** a proposition σ from a sequence of propositions $\sigma_1, \sigma_2, \dots, \sigma_n$. We will now give a short description of such an axiomatic presentation of PL [Schm60, RaSi70, Mend64].

Definition 1.8.1: The Axioms:

We regard as an axiom any proposition of the following form:

- (1) $\varphi \rightarrow (\tau \rightarrow \varphi)$
- (2) $(\varphi \rightarrow (\tau \rightarrow \sigma)) \rightarrow ((\varphi \rightarrow \tau) \rightarrow (\varphi \rightarrow \sigma))$
- (3) $(\neg\varphi \rightarrow \neg\tau) \rightarrow (\tau \rightarrow \varphi)$

Notice that propositions φ, τ and σ can be replaced by just any propositions. We thus have axiomatic schemes (or schemes of axioms), leading to an unlimited number of axioms. One can easily verify that all these axioms are well-formed PL formulae and, of course, tautologies. ■ 1.8.1

Definition 1.8.2: The Modus Ponens rule:

We only use one derivation rule, the rule of **Modus Ponens**, which states that proposition τ can be derived from propositions φ and $\varphi \rightarrow \tau$. Modus Ponens (*mode* according to Diogenes Laertius [Zeno76]) is denoted by:

$$\frac{\varphi \quad \varphi \rightarrow \tau}{\tau} \quad (1)$$

or even with

$$\varphi, \varphi \rightarrow \tau \vdash \tau \quad (2)$$

■ 1.8.2

In (1), which is a characteristic definition of a logical rule, the horizontal line separates the hypothesis from the conclusion. In (2), the symbol “ \vdash ” denotes derivation through the axiomatic system. We consider the three axioms of Definition 1.8.1 as formulae derived in our axiomatic system. New propositions are also produced by means of these three axioms and the Modus Ponens rule. The following example shows how axioms and Modus Ponens can be applied for the derivation of the PL formula $A \rightarrow A$.

Example 1.8.3: Prove that $\vdash A \rightarrow A$.

Proof:

$$\vdash A \rightarrow ((B \rightarrow A) \rightarrow A) \quad (1)$$

by the first axiom. However, by the second axiom we have

$$\vdash (A \rightarrow ((B \rightarrow A) \rightarrow A)) \rightarrow [((A \rightarrow (B \rightarrow A)) \rightarrow (A \rightarrow A))] \quad (2)$$

By (1) and (2) and Modus Ponens, we have

$$\vdash (A \rightarrow (B \rightarrow A)) \rightarrow (A \rightarrow A) \quad (3)$$

But $\vdash A \rightarrow (B \rightarrow A)$ by the first axiom, and then with the Modus Ponens rule, (3) leads to $\vdash A \rightarrow A$. Then proposition $A \rightarrow A$ is derived in our axiomatic system.

■ 1.8.3

The next theorem allows us to replace one of the subformulae of some PL proposition by a logically equivalent one. The proof can be found in Exercise 22.

Theorem 1.8.4: Substitution of equivalences:

If proposition $\sigma \leftrightarrow \sigma_1$ is derived in PL and σ is a subformula of proposition φ , then proposition $\varphi \leftrightarrow \varphi_1$ can also be derived in PL, where φ_1 is the proposition resulting from φ by replacing none, one, or more than one occurrence of σ by its equivalent σ_1 . Formally:

$$\vdash \sigma \leftrightarrow \sigma_1 \Rightarrow \vdash \varphi \leftrightarrow \varphi_1 \quad \blacksquare \text{ 1.8.4}$$

We will now give the formal definition of the proof of a proposition within the axiomatic method.

Definition 1.8.5: Let S be a set of propositions.

- (1) A **proof** from S is a finite sequence of propositions $\sigma_1, \sigma_2, \dots, \sigma_n$, such that for every $1 \leq i \leq n$:
 - (i) σ_i belongs to S , or
 - (ii) σ_i is an axiom, or
 - (iii) σ_i follows from σ_j, σ_k , $1 \leq j, k \leq i$, by application of Modus Ponens.
- (2) A proposition σ is **S -provable** from a set S of propositions, if there is a proof $\sigma_1, \dots, \sigma_n$ from S , such that σ_n coincides with σ . Formally, we write $S \vdash \sigma$.
- (3) Proposition σ is **provable** if $\vdash \sigma$, that is to say if σ is derived in the axiomatic system of Definition 1.8.1 with the use of Modus Ponens. Obviously, the concept of the S -provable proposition coincides with the concept of the provable proposition for $S = \emptyset$. ■ 1.8.5

Example 1.8.6: We give the proof of $\neg B \rightarrow (C \rightarrow A)$ from $S = \{A\}$

- | | |
|--|-----------------------------|
| (1) A | $A \in S$ |
| (2) $A \rightarrow (C \rightarrow A)$ | axiom 1 |
| (3) $(C \rightarrow A)$ | Modus Ponens on (1) and (2) |
| (4) $(C \rightarrow A) \rightarrow (\neg B \rightarrow (C \rightarrow A))$ | axiom 1 |
| (5) $\neg B \rightarrow (C \rightarrow A)$ | Modus Ponens on (3) and (4) |

■ 1.8.6

Note that if proposition σ is provable from S and S is an infinite set, then σ is provable from a finite subset of S , since proofs are always finite.

The theorem below is fundamental in proof theory.

Theorem 1.8.7: Theorem of Deduction:

Let S be a set of propositions and let K, L be two PL propositions. Then:

$$S \cup \{K\} \vdash L \Leftrightarrow S \vdash K \rightarrow L$$

Proof: (\Leftarrow) If $S \vdash K \rightarrow L$ there is a proof of $K \rightarrow L$ denoted by $\sigma_1, \sigma_2, \dots, \sigma_n$, where $\sigma_n \leftrightarrow K \rightarrow L$ and, for every $i \in \{1, \dots, n\}$, σ_i is an axiom, or $\sigma_i \in S$, or σ_i is derived from two preceding propositions by an application of Modus Ponens.

Thus the sequence $\sigma_1, \sigma_2, \dots, \sigma_n, \sigma_{n+1}, \sigma_{n+2}$, where σ_{n+1} is K and σ_{n+2} is L , is a proof of L ; since for every $i \in \{1, 2, \dots, n, n+1\}$, σ_i is an axiom, or $\sigma_i \in S \cup \{K\}$, or σ_{n+2} is derived from σ_n and σ_{n+1} by Modus Ponens.

(\Rightarrow) By the assumption, we know that there is a proof of L from $S \cup \{K\}$ denoted by L_1, L_2, \dots, L_n where L_n is L . Let us consider the following sequence of propositions.

$$K \rightarrow L_1$$

$$K \rightarrow L_2$$

$$\dots$$

$$K \rightarrow L_n$$

This sequence is not a proof. However it becomes a proof if we “add” some propositions between its successive terms using the following inductive method:

For L_1 we have the following cases:

$$L_1 \text{ is an axiom,} \quad \text{or} \quad L_1 \text{ belongs to } S, \quad \text{or} \quad L_1 \text{ is } K$$

By axiom (1) we have $L_1 \rightarrow (K \rightarrow L_1)$. In the two first cases we thus add L_1 and $L_1 \rightarrow (K \rightarrow L_1)$ above $K \rightarrow L_1$. In the last case, the theorem holds true by Example 1.8.3.

Let us assume that we have completed the sequence up to $K \rightarrow L_m$, so that the m first terms of the sequence constitute a proof of $K \rightarrow L_m$.

We will now add ν propositions in the sequence, between propositions $K \rightarrow L_m$ and $K \rightarrow L_{m+1}$, so that the $m+1$ first terms of the sequence, completed by these ν propositions, constitute a proof of $K \rightarrow L_{m+1}$. L_{m+1} is an axiom, or it belongs to S , or it is K , or it is derived from L_j, L_k ($1 \leq j, k \leq m$) by Modus Ponens. In other words, there is an L_k with the form $L_j \rightarrow L_{m+1}$.

For the first three cases, we will work just as we did above, by adding between $K \rightarrow L_m$ and $K \rightarrow L_{m+1}$ the propositions L_{m+1} and $L_{m+1} \rightarrow (K \rightarrow L_{m+1})$. In the fourth case, we know by the inductive assumption that the sequence has been completed so that the m first terms constitute a proof of $K \rightarrow L_j$ and $K \rightarrow (L_j \rightarrow L_{m+1})$. We thus “add” proposition

$$(K \rightarrow (L_j \rightarrow L_{m+1})) \rightarrow ((K \rightarrow L_j) \rightarrow (K \rightarrow L_{m+1}))$$

which is legitimate since the last proposition is axiom (2).

$K \rightarrow L_{m+1}$ is thus derived using two successive applications of Modus Ponens. Note that we have only used two of the three axioms of the axiomatic system to prove the deduction theorem. ■ 1.8.7

The axioms of PL are often called logical axioms. We usually define a theory extending the axiomatization of PL by a set S of additional axioms which characterize the theory. The theorems of the theory S are the elements of the set $\{\varphi \mid S \vdash \varphi\}$ (see Remark 1.8.8).

Remark 1.8.8:

- (1) *The axiomatic system:* The axioms of PL and the Modus Ponens rule constitute the axiomatic system of Frege-Lukasiewicz [Heij67, Boye68, Schm60]. Frege (German, 1898-1925) was the first philosopher and logician to define a formal language suitable for logic. Lukasiewicz (Polish, 1878-1926) dealt with the axiomatization of PL.
- (2) *Modus Ponens:* If σ and $\sigma \rightarrow \tau$ are Beth-provable, then σ and $\sigma \rightarrow \tau$ are logically true, thus τ is also true (why?). There is an algorithmic method which constructs a Beth-proof of τ from the Beth-proofs of σ and $\sigma \rightarrow \tau$. This method is an application of the theorem of Gentzen (Gentzen Hauptsatz), however its proof is beyond the scope of this book.
- (3) *Theorems:* A theorem is any proposition appearing in a proof. This means that the theorems of the theory S are exactly the elements of the set of propositions $\{\sigma \mid S \vdash \sigma\}$. We usually have the conclusion occurring as the last proposition of a proof, but actually every initial part of a proof is also a proof.
- (4) *The selection of axioms:* The axiomatic proof method is sound and complete, as we will show in the following chapters. The axiomatic system chosen is thus complete, meaning that every tautology can be proven from the axioms by successive applications of Modus Ponens.
- (5) *Beth-provability:* Since the axioms themselves are logically true propositions, they are also Beth-provable, and Modus Ponens leads from logically true propositions to propositions which are also logically true. Thus, each theorem is Beth-provable.
- (6) *Axioms and rules:* We could replace one or even more than one of the axioms of the PL axiomatic system by rules. For example, the third axiom

$$(\neg\varphi \rightarrow \neg\tau) \rightarrow (\tau \rightarrow \varphi)$$

could be replaced by the rule

$$\frac{\neg\varphi \rightarrow \neg\tau}{\tau \rightarrow \varphi}$$

The selection between axioms and rules is usually a matter of personal valuation of the specific theoretical requirements.

- (7) *Derivations from axioms:* To prove a proposition from the axioms, we have to try various combinations in order to determine the adequate combination of propositions for the application of Modus Ponens and the axioms. A single derivation, for instance

$$\vdash (A \rightarrow B) \rightarrow ((C \rightarrow A) \rightarrow (C \rightarrow B))$$

thus becomes difficult and time consuming, even with the hint that the first and the second axiom shall both be used twice.

On the contrary, Beth-proofs, as defined in Definition 1.7.6, provide a systematic algorithmic method with a certain result. For this reason we prefer to work with Beth-proofs. ■ 1.8.8

1.9 Resolution

Terminology and Notation in Logic Programming

The resolution method is the most efficient PL algorithmic proof method and, as we will see in the second chapter, for Predicate Logic as well. It constitutes the proof method on which the Logic Programming language PROLOG is based. The resolution method is a proof method by refutation, just like Beth-proofs. It generally has a lot of similarities with the Beth-proof method, but it is more suitable to the writing of logic programs, where the programming language is almost the PL language.

In order to introduce the resolution, we need to define several essential concepts of contemporary Logic Programming.

Definition 1.9.1: A **literal** is any atom or its negation. ■ 1.9.1

For example, $\neg A$, B , $\neg C$ are literals.

We know that we are able to develop any PL proposition into a Conjunctive Normal Form, CNF, which is equivalent to the initial proposition. A CNF is in fact a disjunction of literals, such that in every disjunction no literal occurs more than once.

We now present an algorithm for the construction of a CNF for a given proposition, which operates a lot faster than constructing the proposition's truth table, then selecting the columns, etc..

This algorithm comes as an application of

(i) the laws of De Morgan:

$$\neg(A \wedge B) \leftrightarrow \neg A \vee \neg B \quad \text{and} \quad \neg(A \vee B) \leftrightarrow \neg A \wedge \neg B$$

(ii) the associative properties of \wedge and \vee :

$$(A \wedge B) \wedge C \leftrightarrow A \wedge (B \wedge C) \quad \text{and} \quad (A \vee B) \vee C \leftrightarrow A \vee (B \vee C)$$

(iii) the commutative properties of \wedge and \vee :

$$A \wedge B \leftrightarrow B \wedge A \quad \text{and} \quad A \vee B \leftrightarrow B \vee A$$

(iv) the distributive properties of \wedge over \vee and of \vee over \wedge

$$A \wedge (B \vee C) \leftrightarrow (A \wedge B) \vee (A \wedge C) \quad \text{and} \quad A \vee (B \wedge C) \leftrightarrow (A \vee B) \wedge (A \vee C)$$

(v) the propositions

$$\begin{aligned} A \vee A &\leftrightarrow A, & A \wedge A &\leftrightarrow A, & A \wedge (B \vee \neg B) &\leftrightarrow A, \\ A \vee (B \wedge \neg B) &\leftrightarrow A & \text{and} & & \neg \neg A &\leftrightarrow A \end{aligned}$$

as well as the theorem of substitution of equivalences. (As an exercise, prove that the above propositions are tautologies).

This method will be presented by means of an example.

Example 1.9.2: Develop proposition S into a CNF, where

$$S : \quad \neg((A \vee B) \wedge (\neg A \vee \neg B)) \wedge C$$

Step 1 : We move the negations forward into the parentheses, using the Laws of De Morgan:

$$a : \quad S \leftrightarrow [\neg(A \vee B) \vee \neg(\neg A \vee \neg B)] \wedge C$$

$$b : \quad S \leftrightarrow (\neg A \wedge \neg B) \vee (\neg\neg A \wedge \neg\neg B)] \wedge C$$

Step 2 : We use commutative and associative properties in order to bring together literals of the same atom. We can then simplify double negations, double terms of the kind $A \vee A$ or $A \wedge A$, and superfluous terms of the kind $B \wedge \neg B$ or $B \vee \neg B$, by using the theorem of substitution of equivalences:

$$S \leftrightarrow [(\neg A \wedge \neg B) \vee (A \wedge B)] \wedge C$$

Step 3 : By the distributive properties we have:

$$S \leftrightarrow [((\neg A \wedge \neg B) \vee A) \wedge ((\neg A \wedge \neg B) \vee B)] \wedge C$$

We then continue by repeating the 2nd and 3rd steps until the final CNF is determined.

$$\text{Step } 1' : \quad S \leftrightarrow ((\neg A \wedge \neg B) \vee A) \wedge ((\neg A \wedge \neg B) \vee B) \wedge C$$

$$\text{Step } 3' : \quad \leftrightarrow (\neg A \vee A) \wedge (\neg B \vee A) \wedge (\neg A \vee B) \wedge (\neg B \vee B) \wedge C$$

$$\text{Step } 2' : \quad \leftrightarrow (\neg B \vee A) \wedge (\neg A \vee B) \wedge C$$

which is the CNF of S we are seeking. ■ 1.9.2

The last form of S is a conjunction of literals' disjunctions, and is equivalent to the initial formula. This algorithm generally finishes when the following form of S is determined:

$$(A_1^1 \vee A_2^1 \vee \dots \vee A_k^1) \wedge \dots \wedge (A_1^\nu \vee A_2^\nu \vee \dots \vee A_k^\nu) \quad (*)$$

where the elements of $\{A_1^1, \dots, A_{k_1}^1, \dots, A_1^\nu, \dots, A_k^\nu\}$ are literals.

In the context of the resolution proof method, formulating a proposition as a set of literals proves to be very practical. For instance, the proposition in the first parenthesis in (*) becomes:

$$\{A_1, A_2, \dots, A_{k_1}\}$$

We consider that such a set denotes a disjunction of literals, namely a PL proposition.

We now give the formal definition of the set-theoretical form of a proposition.

Definition 1.9.3: The disjunction of a finite set of *literals* can be *set-theoretically* represented as a set, the elements of which are the considered literals. This set is called a **clause**. A clause is thus equivalent to a PL disjunctive proposition.

For technical reasons, we also introduce the concept of the **empty clause**, a clause which contains no literals and is always *non-verifiable*. An empty clause is denoted by \square . ■ 1.9.3

Definition 1.9.4: The conjunction of a finite set of *clauses* can be *set-theoretically* represented as a set, the elements of which are these clauses. This set is called a **set of clauses**. A set of clauses thus constitutes a conjunction of disjunctions, namely a PL conjunctive proposition. ■ 1.9.4

Example 1.9.5: The set of clauses

$$\underbrace{\{A, B\}}_1, \underbrace{\{\neg B, \neg C\}}_2, \underbrace{\{D\}}_3$$

represents the proposition:

$$\underbrace{((A \vee B))}_1 \wedge \underbrace{(\neg B \vee \neg C)}_2 \wedge \underbrace{D}_3 \quad \blacksquare \quad 1.9.5$$

Remark 1.9.6:

- (1) A truth valuation obviously verifies a set of clauses if it verifies every clause in the set. For example, let $S = \{\{A, B\}, \{\neg C\}\}$ be a set of clauses and let V be a truth valuation such that:

$$V(A) = V(B) = V(C) = t$$

Then V does not verify S , since it does not verify one of its elements, namely $\{\neg C\}$.

- (2) Naturally, we can also consider the **empty set of clauses** $\{\emptyset\}$, which is not to be confused with the empty clause \square . Formally, every truth valuation verifies the empty set of clauses, since it validates every one of its elements (there is no proposition contained in the clauses of $\{\emptyset\}$, see the Proof of Corollary 1.5.4).

On the contrary, each set of clauses which contains the empty clause can be verified by no truth valuation, because \square is not verifiable.

Intuitively, the empty set of clauses denotes that there is *no “claim”* (proposition) for the “world” (the set of propositions), whereas the empty clause denotes that we at least have one proposition for our “world”, the clause \square , which always creates contradictions by making our “world” inconsistent, namely non-verifiable. ■ 1.9.6

In Logic Programming, as well as in most of the PROLOG versions, the above symbolism has prevailed:

Let S be the proposition

$$A_1 \vee \dots \vee A_k \vee (\neg B_1) \vee \dots \vee (\neg B_\ell)$$

where $A_1, \dots, A_k, B_1, \dots, B_\ell$ are atoms. Then we have:

$$\begin{aligned} S &\leftrightarrow A_1 \vee \dots \vee A_k \vee \neg(B_1 \wedge \dots \wedge B_\ell) && \text{by De Morgan} \\ &\leftrightarrow (B_1 \wedge \dots \wedge B_\ell) \rightarrow (A_1 \vee \dots \vee A_k) && \text{by } (\neg B \vee A) \leftrightarrow (B \rightarrow A) \end{aligned}$$

and finally

$$S \leftrightarrow ((A_1 \vee \dots \vee A_k) \leftarrow (B_1 \wedge \dots \wedge B_\ell)) \quad (1)$$

For the use of \leftarrow as a logic connective see also Remark 1.2.9. If now, instead of the logical connectives \wedge, \vee and \leftarrow we wish to use the corresponding symbols “ $,$ ” (comma), “ $;$ ” (semicolon), and “ $:-$ ” (neck symbol), then S can be equivalently denoted by:

$$A_1; \dots; A_k :- B_1, \dots, B_\ell \quad (2)$$

If in proposition (2), $k = 1$ holds, then we have:

$$A_1 :- B_1, \dots, B_\ell \quad (3)$$

Definition 1.9.7: Each clause of form (3) is a **Horn clause**. The atom A is the **head** or the **goal** of S , and the conjunctive components B_1, \dots, B_ℓ , are the **tail** or the **body** or the **subgoals** of S . ■ 1.9.7

The intuitive interpretation of a Horn clause is that, for a goal A to be valid, subgoals B_1, \dots, B_ℓ also need to be valid.

Definition 1.9.8: If $k = 0$ in a (2) form clause, then the clause

$$:- B_1, \dots, B_\ell \quad (4)$$

is called a **program goal** or **definite goal**.

If $\ell = 0$, the clause:

$$A_1 :- \quad (5)$$

is called a **unit clause** or **fact**. ■ 1.9.8

Remark 1.9.9:

- (1) A given set S of clauses can informally be considered as a database, see also section 3.1.1, since the clauses in S represent information about the relationship of the atoms they contain.
- (2) The verification of goal A in the clause $A :- B_1, \dots, B_\ell$ is inferred from the validation of subgoals B_1, \dots, B_ℓ . In such a case, goal A is said to **succeed**, or that there is a proof of A . Otherwise, goal A is said to **fail**.
- (3) The form (4) Horn clause, denoting the absence of a goal, states that at least one of the B_i , $1 \leq i \leq \ell$, fails. The form (5) Horn clause means that A_1 always succeeds. In that case, A_1 constitutes a claim, a fact of our database. ■ 1.9.9

Generally speaking, **resolution** is a deductive rule through which we can derive a proposition in a clause from two other propositions. Before describing formally the method, we shall give an example.

Example 1.9.10: Consider the following clauses

$$\begin{array}{r} \{\neg A, B\} \\ \{A, C\} \\ \hline \{B, C\} \end{array}$$

Using resolution, we can deduce

The intuition in the use of such a rule becomes very clear when we reformulate the previous clauses according to the classical PL formulation.

$$\begin{array}{rcl}
 \text{given propositions} & & (\neg A \vee B) \\
 & & (A \vee C) \\
 \hline
 \text{conclusion} & & (B \vee C)
 \end{array}$$

This rule is an application of the tautology

$$(\neg A \vee B) \wedge (A \vee C) \rightarrow (B \vee C)$$

From the Completeness Theorem 1.10.9, we know that tautologies are derivable by means of the axioms and Modus Ponens. Thus

$$\vdash (\neg A \vee B) \wedge (A \vee C) \rightarrow (B \vee C)$$

Then from the Theorem of Deduction, Theorem 1.8.7, we obtain

$$\{(\neg A \vee B) \wedge (A \vee C)\} \vdash (B \vee C)$$

The rule of resolution is thus derivable in PL. ■ 1.9.10

As a generalization of the previous example, let us consider as given the following clauses:

$$\begin{aligned}
 C_1 &= \{A_1, A_2, \dots, A_{k_1}, \neg B_1, \dots, \neg B_{\ell_1}\} \\
 C_2 &= \{D_1, D_2, \dots, D_{k_2}, \neg F_1, \dots, \neg F_{\ell_2}\}
 \end{aligned}$$

where A_1, \dots, A_{k_1} , B_1, \dots, B_{ℓ_1} , D_1, \dots, D_{k_2} , F_1, \dots, F_{ℓ_2} are atoms. Let us also assume that A_1 coincides with D_1 .

We can then rewrite the two clauses as follows:

$$\begin{aligned}
 C_1 &= \{A_1\} \cup C'_1 \quad \text{where} \quad C'_1 = \{A_2, \dots, A_{k_1}, \neg B_1, \dots, \neg B_{\ell_1}\} \\
 C_2 &= \{\neg A_1\} \cup C'_2 \quad \text{where} \quad C'_2 = \{D_1, \dots, D_{k_2}, \neg F_2, \dots, \neg F_{\ell_2}\}
 \end{aligned}$$

Then the resolution rule that we wish to develop will have to enable us to produce the following clause as a deduction:

$$C = C'_1 \cup C'_2$$

In other words,

$$\begin{array}{rcl}
 \text{given:} & & C_1 = \{A_1\} \cup C'_1 \\
 & & C_2 = \{\neg A_1\} \cup C'_2 \\
 \text{conclusion:} & & C'_1 \cup C'_2 = (C_1 - \{A_1\}) \cup (C_2 - \{\neg A_1\}) \quad (*)
 \end{array}$$

We can consider that the two clauses C_1 and C_2 “collide” because C_1 contains literal A_1 and C_2 literal $\neg A_1$. The removal of the cause of the collision leads to clause $(*)$, which resolves the clash. The method owes its name to this resolution. We can now define formally the resolution method.

Definition 1.9.11: Resolution, a formal description:

Let C_1 and C_2 be two clauses and let L be a literal such that $L \in C_1$ and $(\neg L) \in C_2$. We can then deduce the **resolvent** D of C_1 and C_2 :

$$D = (C_1 - \{L\}) \cup (C_2 - \{\neg L\}) \quad \blacksquare \quad 1.9.11$$

Example 1.9.12:

$$\begin{array}{ll} \text{given:} & C_1 = \{P, Q\} \\ & C_2 = \{\neg P, \neg Q\} \\ \text{conclusion:} & D = \{Q, \neg Q\} \end{array} \quad \blacksquare \quad 1.9.12$$

If we have a set containing more than two clauses, we can introduce the concept of the resolvent set:

Definition 1.9.13: Let $S = \{C_1, C_2, \dots, C_n\}$ be a set of clauses. Then the set $R(S) = S \cup \{D \mid D \text{ is the resolvent of the clauses } C_i, C_j \in S, i \neq j, 1 \leq i, j \leq n\}$ is the **resolvent** of S . $\blacksquare \quad 1.9.13$

Example 1.9.14: Let S be a set of clauses:

$$S = \{\underbrace{\{A, \neg B, \neg C\}}_1, \underbrace{\{B, D\}}_2, \underbrace{\{\neg A, \neg D\}}_3\}$$

By applying the resolution rule on the pairs of clauses of S , we have:

$$\begin{array}{lll} 1 & \{A, \neg B, C\} & 2 \quad \{B, D\} & 3 \quad \{\neg A, \neg D\} \\ 2 & \{B, D\} & 3 \quad \{\neg A, \neg D\} & 1 \quad \{A, \neg B, \neg C\} \\ \hline 4 & \{A, D, \neg C\} & 5 \quad \{B, \neg A\} & 6 \quad \{\neg B, \neg C, \neg D\} \end{array}$$

And finally,

$$R(S) = \{\underbrace{\{A, \neg B, \neg C\}}_1, \underbrace{\{B, D\}}_2, \underbrace{\{\neg A, \neg D\}}_3, \underbrace{\{A, D, \neg C\}}_4, \underbrace{\{B, \neg A\}}_5, \underbrace{\{\neg B, \neg C, \neg D\}}_6\}$$

We can of course continue with the application of the method, by taking successively the following sets: -

$$R^0(S) = S, R^1(S) = R(S), R^2(S) = R(R(S)), \dots, R^n(S) = R(R^{n-1}(S))$$

And finally:

$$R^*(S) = \bigcup_{n=1}^{\infty} R^n(S) = \{Ci \mid Ci \in R^j(S) \text{ and } j \in \mathbb{N}\}$$

where C_i are the clauses contained in the j^{th} resolvent of S .

Note that $R^*(S)$ is a finite set if and only if S is finite. ■ 1.9.14

Remark 1.9.15:

- (1) In Example 1.9.12, we chose to apply resolution through the literal P . We could have done this through Q , since Q is obviously also a cause of collision.
- (2) Intuitively for every resolution application, if a truth valuation verifies C_1 and C_2 then it also verifies their resolvent D . Likewise, whenever a truth valuation verifies S , it also verifies $R(S)$.
- (3) Note that the resolvent D of C_1 captures less information than C_1 and C_2 . This becomes clear with the following example. ■ 1.9.15

Example 1.9.16: Let $S = \{\{A, B\}, \{\neg B\}\}$ be a set of clauses. Then, by resolution, we have

$$\begin{array}{ll} \text{given} & C_1 = \{A, B\} \\ & C_2 = \{\neg B\} \\ & \hline \text{resolution} & D = \{A\} \end{array}$$

By applying resolution on S , we produce $D = \{A\}$ which contains no information about literal B . ■ 1.9.16

We will now give the formal definition of proofs by means of the resolution method.

Definition 1.9.17: Let S a set of clauses. A **resolution proof** from S is a finite sequence of clauses C_1, \dots, C_n , such that, for every C_i , $i = 1, \dots, n$, we have:

$$C_i \in S \quad \text{or} \quad C_i \in R(\{C_j, C_k\}), \quad 1 \leq j, k \leq i \leq n$$

A clause C is **provable by resolution from a set of clauses S** , formally $S \vdash_R C$, if there is a resolution proof from S , the last clause of which is C . Obviously $C \in R^*(S)$. ■ 1.9.17

Example 1.9.18: Find all the resolvents of the set of clauses

$$S = \{\{A, B\}, \{\neg A, \neg B\}\}$$

Let us number all the clauses in S :

1. $\{A, B\}$
2. $\{\neg A, \neg B\}$
3. $\{B, \neg B\}$ from 1 and 2.
4. $\{A, \neg B\}$ from 1 and 3.

Then

$$R^1(S) = \{\{A, B\}, \{\neg A, \neg B\}, \{B, \neg B\}, \{A, \neg A\}\}$$

Finally

$$\begin{aligned} R^*(S) &= R^0(S) \cup R^1(S) \\ &= \{\{A, B\}, \{\neg A, \neg B\}, \{B, \neg B\}, \{A, \neg A\}\} \end{aligned}$$

Clauses of the kind $\{A, \neg A\}$, namely $A \vee \neg A$, are tautologies. ■ 1.9.18

Example 1.9.19: The following proposition is given:

$$S : ((A \leftrightarrow (B \rightarrow C)) \wedge (A \leftrightarrow B) \wedge (A \leftrightarrow \neg C))$$

Prove that S is not verifiable.

Proof:

Step 1: Determine the CNF of S

$$\begin{aligned}
 S &\leftrightarrow ((A \rightarrow (B \rightarrow C)) \wedge ((B \rightarrow C) \rightarrow A) \wedge (A \rightarrow B) \\
 &\quad \wedge (B \rightarrow A) \wedge (A \rightarrow \neg C) \wedge (\neg C \rightarrow A)) \\
 &\leftrightarrow \underbrace{(\neg A \vee \neg B \vee C)}_1 \wedge \underbrace{(B \vee A)}_2 \wedge \underbrace{(\neg C \vee A)}_3 \wedge \underbrace{(\neg A \vee B)}_4 \\
 &\quad \wedge \underbrace{(\neg B \vee A)}_5 \wedge \underbrace{(\neg A \vee \neg B)}_6 \wedge \underbrace{(C \vee \neg A)}_7
 \end{aligned}$$

Step 2: Form the corresponding set of clauses:

$$S = \{ \underbrace{\{\neg A, \neg B, C\}}_1, \underbrace{\{B, A\}}_2, \underbrace{\{\neg C, A\}}_3, \underbrace{\{\neg A, B\}}_4, \underbrace{\{\neg B, A\}}_5, \underbrace{\{\neg A, \neg C\}}_6, \underbrace{\{C, A\}}_7 \}$$

Step 3: Determine the various resolvents:

8.	$\{A\}$	by 2 and 5.
9.	$\{\neg A, \neg B\}$	by 2 and 6.
10.	$\{\neg A\}$	by 4 and 9.
11.	\square	by 8 and 10.

(see also Definition 1.9.8. The literal $\neg A$ is eliminated and clause 11 contains no literals).

Since the empty clause belongs to resolvent by 11, the set of clauses S is not verifiable. Thus, proposition S is not verifiable. ■ 1.9.19

Example 1.9.20: Prove that the proposition $\neg B$ is provable by resolution from the set

$$Q = \{ \{A, \neg B\}, \{\neg A, \neg B, \neg C\}, \{\neg A, \neg B, C\} \}$$

Proof:

1. $\{A, \neg B\}$
2. $\{\neg A, \neg B, \neg C\}$
3. $\{\neg A, \neg B, C\}$
4. $\{\neg A, \neg B\}$ by 2 and 3.
5. $\{\neg B\}$ by 4 and 1.

The resolution proof we are seeking is the sequence of clauses 1, 2, 3, 4 and 5.

■ 1.9.20

Remark 1.9.21: The proof in Example 1.9.20 could also have been conducted as follows:

We apply resolution on

$$S_1 = \underbrace{\{A, \neg B\}}_1, \underbrace{\{\neg, A, \neg B, \neg C\}}_2, \underbrace{\{\neg A, \neg B, C\}}_3, \underbrace{\{B\}}_4 = S \cup \{B\}$$

to give

- | | | |
|----|----------------------|-------------|
| 5. | $\{A\}$ | by 1 and 4. |
| 6. | $\{\neg A, \neg B\}$ | by 2 and 3. |
| 7. | $\{\neg, A\}$ | by 4 and 6. |
| 8. | \square | by 5 and 7. |

Namely we have $S \cup \{B\} \vdash_R \square$. Since the rule of resolution is a derivable rule of PL as we saw in Example 1.9.10, we have also $S \cup \{B\} \vdash \square$. But in that case, we have, by the Deduction Theorem 1.8.6, that $S \vdash B \rightarrow \square$.

By the tautology $(B \rightarrow \square) \leftrightarrow \neg B$, we can conclude $S \vdash \neg B$; in other words $\neg B$ is provable from S . ■ 1.9.21

1.10 Soundness and Completeness of Tableaux

In the next sections we will give the basic theorems on soundness and completeness of the proof methods we have presented. We will start with soundness and completeness of Beth-proofs.

For each of the following definitions or theorems which contain the notions “Beth-proof” and “logically true”, there are dual definitions and theorems corresponding to the dual notions “Beth-refutation” and “logically false”, respectively. The formulation of those definitions and theorems is left as an exercise to the reader.

We will prove that all Beth-provable propositions are true (soundness), and conversely, that every logically true proposition is Beth-provable (completeness) [Smul68]. The proofs which we will present are inductive on the length of a proposition, or on the length of a semantic tableau. We have already described the induction scheme for propositions. We will now describe the general induction scheme for semantic tableaux.

Definition 1.10.1: Induction Scheme for Semantic Tableaux:

Let P be a property of some semantic tableaux T , symbolically $P(T)$. If we prove that:

- (a) every atomic semantic tableau has property P
- (b) if P is a property of a semantic tableau T , and if T' is a new semantic tableau formed by the concatenation of an atomic semantic tableau at the end of one of T 's branches, then P is also a property of T'

we can then deduce that P is a property of all semantic tableaux, namely that $P(T)$ holds *for every semantic tableau* T . ■ 1.10.1

For a semantic tableau T , the induction is carried out on the **length** of T , namely the number of atomic semantic tableaux in T .

The analogy between the induction scheme for propositions, Definition 1.2.2, and the induction scheme for semantic tableaux is obvious.

Example 1.10.2: Let P be the property “The number of nodes of a semantic tableau is greater than or equal to the number of branches”.

Proof:

- (a) In all atomic semantic tableaux, the number of nodes is greater than or equal to the number of branches.
- (b) Let us assume that in the semantic tableau T , the number of nodes is greater than or equal to the number of branches. We proceed by concatenating at the end of one branch of T an atomic semantic tableau. If the concatenation node is $t(\neg\sigma)$ or $f(\neg\sigma)$, then we have one more node while the number of branches remains the same. For any other type of concatenation node, there will be no more than two new branches and there will be at least two new nodes, according to Definition 1.7.1. Thus, in the new semantic tableau T' , the number of nodes is once again greater than or equal to the number of branches. ■ 1.10.2

We now give several auxiliary definitions and lemmata concerning soundness and completeness theorems for Beth proofs.

Definition 1.10.3: Let κ be a branch of a semantic tableau T , and let $P = \{P_1, \dots, P_n\}$ be the set of nodes of κ , where for each proposition σ , either $P_i = t\sigma$ or $P_i = f\sigma$. Then the **truth valuation V agrees with branch κ** if for every $P_i \in P$:

$$P_i = t\sigma \Rightarrow V(\sigma) = t \quad \text{and} \quad P_i = f\sigma \Rightarrow V(\sigma) = f \quad \blacksquare \quad 1.10.3$$

Lemma 1.10.4: *Let V be a truth valuation which agrees with the origin of a semantic tableau, which means that if the origin is $t\sigma$, then $V(\sigma) = t$, and if the origin is $f\sigma$, then $V(\sigma) = f$. Then V agrees with a branch of the semantic tableau.*

Proof: By induction:

- (a) The lemma obviously holds for atomic semantic tableaux.
- (b) Let us assume that the lemma holds for a semantic tableau T . Let T' be the semantic tableau built by concatenating an atomic semantic tableau with an origin X at the end of a branch κ of T . We wish to prove now that the lemma holds for the semantic tableau T' .

Case 1: V agrees with all nodes in branch κ . Then V agrees with the node X and thus agrees with one of the branches of the atomic semantic tableau with X at the origin. Let κ_1 be the agreement branch of V and the atomic semantic tableau. The branch $\kappa\kappa_1$ resulting from the concatenation of κ and κ_1 is the branch of T we are looking for and which V agrees with.

Case 2: V does not agree with κ but it agrees with the origin of T , otherwise we need not prove anything. Then, there is another branch κ' of T which V agrees with. But κ' is also a branch of T . Hence V agrees with κ' . $\blacksquare \quad 1.10.4$

Lemma 1.10.5: Hintikka's Lemma:

Let κ be a non-contradictory branch of a complete semantic tableau. We define a truth assignment, and thus the corresponding truth valuation, as follows:

$$\begin{aligned} V(A) &= t && \text{if } tA \text{ is a node of } \kappa \\ V(A) &= f && \text{if } tA \text{ is not a node of } \kappa \end{aligned}$$

Then V agrees with branch κ .

Proof: By induction on the length of propositions:

- (a) If A is an atom and tA is a node of κ , then $V(A) = t$ and V agrees with κ .
If fA is a node, since κ is not contradictory, tA is not a node and hence $V(A) = f$.
- (b) If $t(\sigma_1 \wedge \sigma_2)$ is a node of κ , then, since the semantic tableau is complete, this node was used at some point and

$$\begin{array}{c} t\sigma_1 \\ | \\ t\sigma_2 \end{array}$$

has been concatenated at the end of branch κ .

Hence, $t\sigma_1$ and $t\sigma_2$ are nodes of branch κ . According to the hypothesis, $V(\sigma_1) = t$ and $V(\sigma_2) = t$ and so $V(\sigma_1 \wedge \sigma_2) = t$. If $f(\sigma_1 \wedge \sigma_2)$ is a node of κ , then, since the semantic tableau is complete, the node $f(\sigma_1 \wedge \sigma_2)$ was used at some point and

$$\begin{array}{cc} & \diagdown \quad \diagup \\ f\sigma_1 & f\sigma_2 \end{array}$$

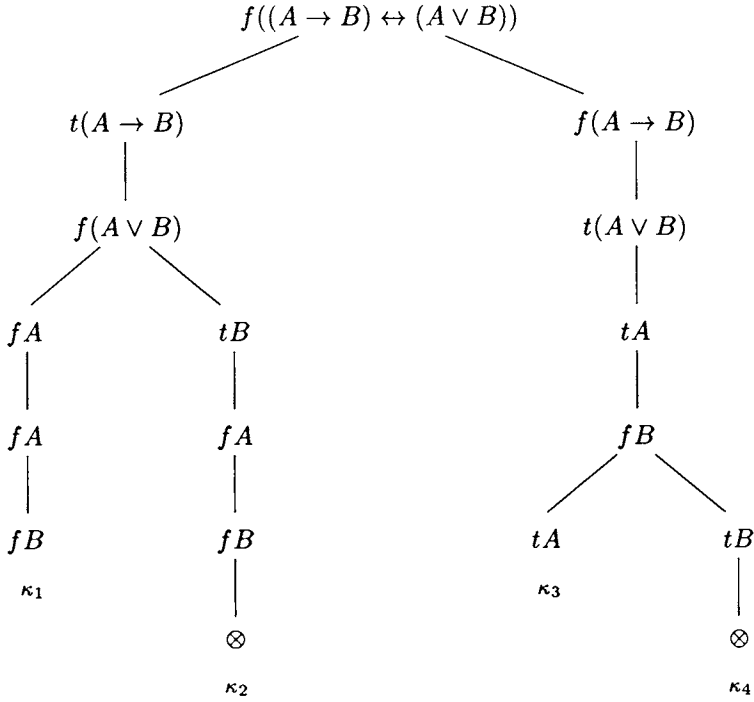
has been concatenated at the end of branch κ . But in that case, either $f(\sigma_1)$ or $f(\sigma_2)$ are nodes of κ . And therefore, according to the hypothesis, $V(\sigma_1) = f$ or $V(\sigma_2) = f$, and hence $V(\sigma_1 \wedge \sigma_2) = f$.

The remaining possible cases analysed in the same way are left as exercises for the reader. ■ 1.10.5

Hintikka's lemma provides us in practice with an algorithm for the construction of a counterexample to the claim that a proposition is logically true. Let us assume we are given a proposition σ . We then build a complete semantic tableau with an $f\sigma$ origin. If the semantic tableau is contradictory, then the proposition is indeed logically true. If the semantic tableau is not contradictory, then it has at least one non-contradictory branch κ . Hintikka's lemma indicates to us how to construct a truth valuation such that $V(\sigma) = f$ based on κ .

Let us examine this procedure with an example.

Example 1.10.6: Find a truth valuation V such that $V((A \rightarrow B) \leftrightarrow (A \vee B)) = f$



Branches κ_2 and κ_4 are contradictory. We can use any of the non-contradictory branches κ_1 and κ_3 to apply Hintikka's lemma. We have, for example, truth valuations V_1 and V_3 , with:

$$V_1(A) = f, \quad V_1(B) = f \quad \text{and} \quad V_3(A) = t, \quad V_3(B) = f$$

such that:

$$V_1((A \rightarrow B) \leftrightarrow (A \vee B)) = V_3((A \rightarrow B) \leftrightarrow (A \vee B)) = f$$

This means that we have found two truth valuations giving the proposition $(A \rightarrow B) \leftrightarrow (A \vee B)$ truth value f . ■ 1.10.6

Theorem 1.10.7: Soundness Theorem:

If a proposition σ is Beth-provable, then it is also logically true. Formally:

$$\vdash_B \sigma \Rightarrow \models \sigma$$

Proof: If proposition σ is not logically true, then there is a truth valuation V such that $V(\sigma) = f$. According to Lemma 1.10.4, every semantic tableau with $f\sigma$ at the origin has at least one branch κ which agrees with V , and consequently κ is not contradictory (why?). Thus σ is not Beth-provable. ■ 1.10.7

Remark 1.10.8: In the proof of Theorem 1.10.7 we used in our metalanguage the third Axiom

$$(\neg\mathcal{P}_1 \Rightarrow \neg\mathcal{P}_2) \Rightarrow (\mathcal{P}_2 \Rightarrow \mathcal{P}_1).$$

Instead of proving $\mathcal{P}_2 \Rightarrow \mathcal{P}_1$, i.e., if σ is Beth-provable then σ is logically true, we proved that $\neg\mathcal{P}_1 \Rightarrow \neg\mathcal{P}_2$, in other words if σ is not logically true then σ is not Beth-provable. This proof method is used quite often and is called an **indirect** or **contrapositive** proof. (The direct proof method would be to prove $\mathcal{P}_2 \Rightarrow \mathcal{P}_1$ directly.) ■ 1.10.8

Theorem 1.10.9: Completeness Theorem:

If a proposition σ is logically true, then it is also Beth-provable. Formally:

$$\models \sigma \Rightarrow \vdash_B \sigma$$

Proof: If the proposition σ is logically true, then for every truth valuation V , $V(\sigma) = t$ holds. Let us assume there is no Beth-proof for σ . We construct a complete semantic tableau with an $f\sigma$ origin. This semantic tableau must have a non-contradictory branch. According to Lemma 1.10.4, there is an adequately defined truth valuation V which agrees with this branch and, hence, with the $f\sigma$ origin. But then $V(\sigma) = f$, a contradiction. Hence, there is a Beth-proof for σ . ■ 1.10.9

It is obvious from the completeness proof that if we tried to construct a Beth-proof for a proposition σ , (namely a complete semantic tableau with $f\sigma$ at its origin) and failed, that is, when the constructed complete semantic tableau has at least one non-contradictory branch, then, exactly as in Lemma 1.10.5, a truth valuation can be defined which is a counterexample to the claim that σ is logically true. In other words *by constructing a semantic tableau of σ we are guaranteed either to obtain a proof of σ or a counterexample to the claim that σ is true.*

1.11 Deductions from Assumptions

The Compactness Theorem

Automatic theorem proving from assumptions and data is an important objective in Logic Programming. In section 1.5, we discussed the consequences of a set of assumptions S . A proposition σ was called a consequence of S , $S \models \sigma$, if every truth valuation which makes all the propositions in S valid also assigns value t to σ . We can now define what derivation of a proposition from a set of propositions and assumptions means.

Definition 1.11.1: Let

$$\sigma, \varphi_1, \varphi_2, \dots, \varphi_n, \dots$$

be a finite or infinite sequence of propositions. σ is said to be a **Beth-deduction** of $\varphi_1, \varphi_2, \dots, \varphi_n, \dots$ if there is a contradictory semantic tableau constructed as follows:

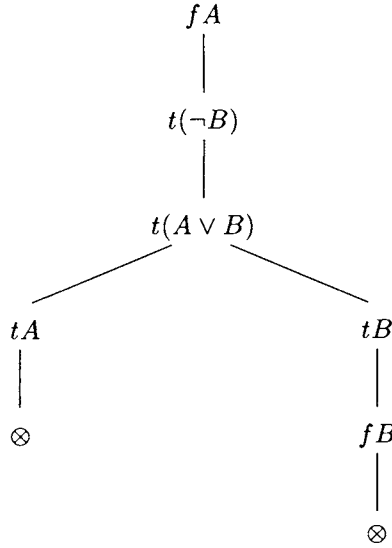
- Step 0:* we start with $f\sigma$ as the origin.
- Step S_{2n} :* we place $t\varphi_n$ at the end of every non-contradictory branch.
- Step S_{2n+1} :* we apply the development rules on the semantic tableau T_{2n} of the previous step.

■ 1.11.1

If the sequence of propositions is infinite, this construction may never finish. σ is a Beth-deduction only if the construction finishes and if a contradictory semantic tableau is built. Then intuitively, there is no truth valuation which assigns truth value t to all propositions φ_n used in the construction of the semantic tableau, and which also assigns truth value f to σ . If the initial sequence is finite, then the construction will certainly finish, yielding a complete semantic tableau.

Let us illustrate the above construction by means of an example.

Example 1.11.2: We wish to prove that proposition A is a Beth-deduction of propositions $\neg B$ and $(A \vee B)$. We start with fA and we concatenate successively the assertions $t(\neg B)$ and $t(A \vee B)$, as in the tableau on the facing page.



We examine $t(A \vee B)$. The left branch is contradictory and need not be examined further. In the right branch we examine $t(\neg B)$. This branch is also contradictory. Thus A is a Beth-deduction of $\neg B$, $(A \vee B)$. ■ 1.11.2

The two following theorems refer to finite sets of assumptions. They correspond to the soundness and completeness theorems of the previous section.

Theorem 1.11.3: Soundness of Deductions:

If proposition σ is a Beth-deduction of $\varphi_1, \varphi_2, \dots, \varphi_n$, then σ is a consequence of $\varphi_1, \varphi_2, \dots, \varphi_n$. Formally:

$$\{\varphi_1, \varphi_2, \dots, \varphi_n\} \vdash_B \sigma \Rightarrow \{\varphi_1, \varphi_2, \dots, \varphi_n\} \models \sigma$$

Proof: Indirectly:

Suppose σ is not a consequence of $\varphi_1, \dots, \varphi_n$. Then there is a truth valuation V with the property

$$V(\varphi_1) = \dots = V(\varphi_n) = t$$

while $V(\sigma) = f$. From Lemma 1.10.4, each semantic tableau with an $f\sigma$ origin has at least one branch which agrees with V , and so is non-contradictory. Thus all semantic tableaux with an $f\sigma$ origin are non-contradictory. Then σ cannot be Beth-provable from $\{\varphi_1, \dots, \varphi_n\}$. ■ 1.11.3

Theorem 1.11.4: Completeness of Deductions:

If a proposition σ is a consequence of $\varphi_1, \varphi_2, \dots, \varphi_n$ then σ is a Beth-deduction of $\varphi_1, \varphi_2, \dots, \varphi_n$. Formally:

$$\{\varphi_1, \varphi_2, \dots, \varphi_n\} \models \sigma \Rightarrow \{\varphi_1, \varphi_2, \dots, \varphi_n\} \vdash_B \sigma$$

Proof: Let us assume that σ is not a Beth-deduction of $\varphi_1, \varphi_2, \dots, \varphi_n$. Then we can construct a complete non-contradictory semantic tableau with an $f\sigma$ origin, every branch of which contains the nodes $t\varphi_1, t\varphi_2, \dots, t\varphi_n$. There thus exists a branch of this semantic tableau which is non-contradictory, and according to Lemma 1.10.5 there is a truth valuation V which agrees with this branch. Hence we have $V(\varphi_1) = V(\varphi_2) = \dots = V(\varphi_n) = t$ and $V(\sigma) = f$. However, this is a contradiction, since σ is a consequence of $\varphi_1, \varphi_2, \dots, \varphi_n$. ■ 1.11.4

Remark 1.11.5: The Soundness and Completeness Theorems 1.11.3 and 1.11.4, respectively, hold even if the sequence $\{\varphi_k, k \in \mathbb{N}\}$ has an infinite number of terms. ■ 1.11.5

We can now formulate, prove and apply the PL compactness theorem. First we will give a basic definition.

Definition 1.11.6: A sequence $\sigma_1, \sigma_2, \dots, \sigma_n$ is called **satisfiable** if there is a truth valuation V such that $V(\sigma_1) = V(\sigma_2) = \dots = V(\sigma_n) = t$. V is then said to **satisfy** the sequence $\sigma_1, \sigma_2, \dots, \sigma_n$. ■ 1.11.6

Example 1.11.7: If A_1, A_2, \dots is a sequence of atoms, then the infinite sequence $A_1, A_2, A_1 \wedge A_2, A_3, A_1 \wedge A_3, A_2 \wedge A_3, \dots$ is satisfiable. (A truth valuation V such that $t = V(A_1) = V(A_2) = V(A_3) = \dots$ satisfies the sequence). On the contrary, $A_1, A_2, (A_1 \rightarrow A_3), (\neg A_3)$ is a finite sequence which is not satisfiable. Indeed, if we assume that it is satisfiable, then there is a truth valuation V such that:

$$V(A_1) = V(A_2) = V(A_1 \rightarrow A_3) = V(\neg A_3) = t$$

In other words, $V(A_3) = f$ while $V(A_1) \rightsquigarrow V(A_3) = t$, which means that $V(A_1) = f$, and that is a contradiction. ■ 1.11.7

Before we formulate the compactness theorem, we will give a definition and a lemma which are necessary for the corresponding proof.

Definition 1.11.8:

- (1) Let X, Y be two nodes of a semantic tableau. Y is a **descendant** of X if there is a branch passing through X and Y , and X is closer to the origin than Y . Y is a **immediate descendant** of X if Y is a descendant of X and there is no other node between X and Y in the branch passing through X and Y . X is said to be **suitable** if it has an infinite number of descendants.
- (2) A semantic tableau is said to be of a **finite degree** if each of its nodes has only *finitely* many immediate descendants. ■ 1.11.8

Lemma 1.11.9: Koenig's Lemma:

A semantic tableau of finite degree with an infinite number of nodes has at least one infinite branch.

Proof: Note that the origin of a semantic tableau with infinitely many nodes is a suitable node. Furthermore, if X is a suitable node, then at least one of its immediate descendants is also a suitable node, since the semantic tableau is of finite degree. Thus, if X_0 is the node of the origin, if X_1 is an immediate descendant of X_0 and a suitable node as well, if X_2 is an immediate descendant of X_1 and a suitable node as well, if \dots , and so on, then the branch $X_0X_1\dots$ has an infinite number of nodes. ■ 1.11.9

Theorem 1.11.10: Compactness Theorem:

Let $\sigma_1, \sigma_2, \dots$ be an infinite sequence of propositions. If for every n , the finite sequence $\sigma_1, \sigma_2, \dots, \sigma_n$ is satisfiable, then the sequence $\sigma_1, \sigma_2, \dots$ is satisfiable.

Proof: We will describe inductively the construction of a semantic tableau which may be infinite.

Step 1: Start with $f(\neg\sigma_1)$ at the origin.

Step 2n: Place $t(\sigma_n)$ at the end of every non-contradictory branch of the T_{2n-1} tableau constructing thus tableau T_{2n} .

Step 2n + 1: Select the non-used node of T_{2n} which is furthest left and apply the development rules of Definition 1.7.1.

Let us suppose that the construction finishes if, at an odd step $2n + 1$, all branches are contradictory. (If they are not contradictory, we have to continue the construction with the next step $2n + 1$). If that happens, we have a contradictory semantic tableau with origin $f(\neg\sigma_1)$ and with $\sigma_1, \sigma_2, \dots, \sigma_n$ as hypotheses. By Theorem 1.11.3, we know that $\neg\sigma_1$ is a consequence of $\sigma_2, \sigma_3, \dots, \sigma_n$, and hence $\sigma_1, \dots, \sigma_n$ is not satisfiable, which contradicts the assumptions.

The construction thus never finishes, and so results in a semantic tableau with an infinite number of nodes. This semantic tableau is of finite degree, since each one of its nodes has finitely many direct descendants. By applying Koenig's Lemma, we thus have a semantic tableau with one infinite branch κ , where every $t\sigma_i$ is a node of κ for every $i = 1, 2, \dots$

We now define a truth valuation V such that:

$$V(A) = t \Leftrightarrow A \text{ is a propositional symbol and } tA \text{ is a node of } \kappa.$$

Then from Hintikka's lemma we can deduce that $V(\sigma_i) = t$ for every i , and $\sigma_1, \sigma_2, \dots, \sigma_n$ is satisfiable. ■ 1.11.10

1.12 Soundness and Completeness of Axiomatic Proofs

We now present the soundness, completeness and compactness theorems of the axiomatic method. The proofs of these theorems are not given here, but the reader can find them in most classical logic books, such as [Klee52, Rasi74, RaSi70].

Theorem 1.12.1: *σ is provable from a set S of propositions if and only if σ is a consequence of S . Formally:*

$$S \vdash \sigma \Leftrightarrow S \models \sigma \quad \text{■ 1.12.1}$$

Corollary 1.12.2: *σ is provable from $S = \emptyset \Leftrightarrow \sigma$ is logically true.* ■ 1.12.2

Theorem 1.12.3: Compactness:

S is a satisfiable set of propositions if and only if every finite subset of S is satisfiable. ■ 1.12.3

1.13 Soundness and Completeness of Resolution

In the following, we will deal with the soundness and completeness theorems of the resolution method.

Theorem 1.13.1: Soundness and Completeness of Resolution:

S is a non-verifiable set of clauses if and only if $R^(S)$ contains the empty clause. Formally:*

$$\square \in R^*(S) \Leftrightarrow S \text{ is non-verifiable.} \quad \blacksquare \text{ 1.13.1}$$

For the proof of the soundness theorem we first need to prove an auxiliary lemma.

Lemma 1.13.2: *If $\{C_1, C_2\}$ is a verifiable set of clauses and if C is the resolvent of C_1 and C_2 , then C is verifiable.*

Proof: For some literal p , we have $C_1 = \{p\} \cup C'_1$, $C_2 = \{\neg p\} \cup C'_2$, and $C = C'_1 \cup C'_2$. If V is a truth valuation verifying $\{C_1, C_2\}$, then $V(p) = t$ or $V(\neg p) = t$. Let us suppose $V(p) = t$. Since $V(C_2) = t$ and $V(\neg p) = f$, then $V(C'_2) = t$, and so $V(C) = t$. If $V(\neg p) = t$, we simply replace C_2 with C_1 .

\blacksquare 1.13.2

Theorem 1.13.3: Soundness of Resolution:

If $\square \in R^(S)$, then S is non-verifiable. Formally:*

$$\square \in R^*(S) \Rightarrow S \text{ non-verifiable.}$$

Proof: Let C_1, \dots, C_k be a proof from S by resolution. Then with the inductive use of the above lemma, we prove that any truth valuation verifying S also verifies C_1 . If the conclusion is the empty clause, then C_k coincides with \square . Since the empty clause \square is non-verifiable, S is non-verifiable. \blacksquare 1.13.3

To prove the completeness of resolution, we will use the following auxiliary lemma.

Lemma 1.13.4: *Let S be a non-verifiable set of clauses in which only the literals A_1, A_2, \dots, A_k occur. Let S^{k-1} be the finite set of clauses which are provable by resolution, and in which the only atomic propositions occurring are A_1, A_2, \dots, A_{k-1} . Then S^{k-1} is non-verifiable.*

Proof: Let us suppose that S^{k-1} is verifiable. Then there is a truth valuation V on $\{A_1, \dots, A_{k-1}\}$ which verifies S^{k-1} . Let V_1 and V_2 be the two extensions of V on $\{A_1, \dots, A_k\}$ such that:

$$V_1(A_k) = t \quad \text{and} \quad V_2(A_k) = f$$

Since S is non-verifiable, there is a clause $C_1 \in S$ which is not verifiable by V_1 . However, then $\neg A_k \in C_1$, since otherwise there are two possibilities:

- (a) $A_k \notin C_1$. But then, V_1 does not contain A_k , $C_1 \in S^{k-1}$ and V_1 verifies C_1 , contradiction.
- (b) $A_k \in C_1$. Then V_1 verifies S^{k-1} and $V_1(A_k) = t$. V_1 therefore verifies C_1 , contradiction.

We can also conclude, in a similar way, that there is a clause $C_2 \in S$, which is non-verifiable by V_2 , such that $A_k \in C_2$.

Let us consider $D = (C_1 - \{\neg A_k\}) \cup (C_2 - \{A_k\})$. D is the resolvent of C_1, C_2 and $D \in S^{k-1}$. Hence, V verifies D . In other words, one of the following holds:

- (a') V verifies $C_1 - \{\neg A_k\}$. But then V_1 verifies C_1 , contradiction.
- (b') V verifies $C_2 - \{A_k\}$. But then V_2 verifies C_2 , contradiction.

Then S^{k-1} is non-verifiable. ■ 1.13.4

Theorem 1.13.5: Completeness of Resolution:

If S is a non-verifiable set of clauses, then the empty clause is provable by resolution from S . Formally:

$$S \text{ non-verifiable} \Rightarrow \square \in R^*(S).$$

Proof: By Definition 1.9.3, S contains a finite number of literals. Let A_1, \dots, A_k be these literals. By applying the previous lemma once, we conclude that S^{k-1} is non-verifiable. Applying the same lemma k times we conclude that S is non-verifiable, since no atomic propositions occur in its clauses and $S^0 \subseteq R^*(S)$. Hence, $\square \in S^0 \subseteq R^*(S)$. ■ 1.13.5

Remark 1.13.6: If σ is a proposition provable by resolution from the set S of propositions, then σ is provable from S (Definition 1.8.5). We then have by the

Completeness Theorem 1.12.1

$$S \vdash_R \sigma \Leftrightarrow S \vdash \sigma \Leftrightarrow S \models \sigma \quad (1)$$

By the Soundness and Completeness Theorems 1.10.6 and 1.10.8, (1) can be generalized:

$$S \vdash_R \sigma \Leftrightarrow S \vdash_B \sigma \Leftrightarrow S \vdash \sigma \Leftrightarrow S \models \sigma \quad \blacksquare \quad 1.13.6$$

1.14 Exercises

1.14.1

Prove that, according to the definition of propositions, the following expressions are propositions:

- (a) $p : (A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C))$
- (b) $r : ((A \vee B) \rightarrow C) \rightarrow (\neg A \wedge C)$
- (c) $s : ((A_1 \wedge A_2) \vee (\neg A_3))$

Solution:

- (a) We will prove that p is a proposition by constructing p inductively according to Definition 1.2.1.
 - (1) A, B, C are propositions, by the definition of the PL language.
 - (2) $(B \wedge C)$ is a proposition by Definition 1.2.1 (ii) and (1).
 - (3) $(A \vee (B \wedge C))$ is a proposition by Definition 1.2.1 (ii), and (1), (2).
 - (4) $(A \vee B)$ and $(A \vee C)$ are propositions by Definition 1.2.1 (ii) and (1).
 - (5) $(A \vee B) \wedge (A \vee C)$ is a proposition by Definition 1.2.1 (ii) and (4).
 - (6) $p : (A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C))$ is a proposition by Definition 1.2.1 (ii) and (5).

The same method can be used to solve (b) and (c).

1.14.2

Determine which of the following expressions are propositions and which are not:

- | | |
|-------------------------------------|---|
| (a) $(A \wedge B) \vee \neg$ | (b) $((A \wedge B) \vee (\neg C)) \rightarrow D$ |
| (c) $(A \vee B) \vee \rightarrow C$ | (d) $A \leftrightarrow B) \rightarrow (A \vee B)$ |
| (e) $(A \wedge B) \rightarrow A$ | (f) $(A_1 \wedge A_2) \leftrightarrow \neg A_3$ |

Solution:

- (a) It is not a proposition. \vee is followed by a logical connective without any propositional symbol.
- (b) It is a proposition according to Definition 1.2.1 (proof: see exercise 1).
- (c) It is not a proposition. \vee is followed by a connective and not by a proposition.
- (d) It is not a proposition. There is a defective use of parentheses.
- (e) It is a proposition according to Definition 1.2.1 (proof: see exercise 1).
- (f) It is a proposition according to Definition 1.2.1 (proof: see exercise 1).

1.14.3

Represent the following propositions by atomic symbols and use those symbols to create compound propositions.

- (a) "2 divides 12"
"3 divides 9"
"2 divides 11"
- (b) "a square is a parallelogram"
"a rhombus is a parallelogram"
"the diagonals of a parallelogram bisect"
- (c) "George is a father"
"George has a child"
"Mary is a father"
"Mary has a child"

Solution:

- (a) Consider
- $$\begin{aligned} A &: \text{“2 divides 12”} \\ B &: \text{“3 divides 9”} \\ C &: \text{“2 divides 11”} \end{aligned}$$

$A \vee B$, $A \wedge B \rightarrow C$, $(A \vee B) \wedge (\neg C)$, and $A \rightarrow (B \wedge \neg C)$ are examples of compound propositions using the symbols A, B and C . $(A \wedge B \rightarrow C)$ is a proposition; however, intuitively, the statement of that proposition does not seem correct.

- (b) and (c) can be solved similarly.

1.14.4

The following atomic propositions are given:

- $$\begin{aligned} A_1 &: \text{“3 is a prime number”} & A_2 &: \text{“3 divides 15”} \\ A_3 &: \text{“3 divides 2”} & A_4 &: \text{“3 divides 13”} \end{aligned}$$

- (a) Determine a valuation F for the above propositions.
 (b) Let W_F be the truth valuation extending F . Calculate

$$W_F((A_1 \wedge A_2) \rightarrow (A_3 \vee A_4))$$

Solution:

- (a) Let F be a valuation such that:

$$\begin{aligned} F(A_1) &= f & F(A_2) &= t \\ F(A_3) &= f & F(A_4) &= t \end{aligned}$$

- (b) $W_F[(A_1 \wedge A_2) \rightarrow (A_3 \vee A_4)]$

$$\begin{aligned} &= W_F(A_1 \wedge A_2) \rightsquigarrow W_F(A_3 \vee A_4) \\ &= [W_F(A_1) \sqcap W_F(A_2)] \rightsquigarrow [W_F(A_3) \sqcup W_F(A_4)] \\ &= [F(A_1) \sqcap F(A_2)] \rightsquigarrow [F(A_3) \sqcup F(A_4)] \\ &= (f \sqcap t) \rightsquigarrow (f \sqcup t) \\ &= f \rightsquigarrow t = t \end{aligned}$$

1.14.5

Prove that the following propositions are tautologies:

- (a) $(A \wedge \neg A) \rightarrow A$ (b) $(A \rightarrow B) \vee (A \rightarrow \neg B)$
 (c) $A \rightarrow \neg \neg A$ (d) $[(A \wedge B) \rightarrow C] \leftrightarrow [A \rightarrow (B \rightarrow C)]$

Solution:

- (a) Consider the valuation $W(A) = t$, and the truth valuation W' extending W . Then:

$$\begin{aligned}
 W'[(A \wedge \neg A) \rightarrow A] &= W'(A \wedge \neg A) \rightsquigarrow W'(A) \\
 &= [W'(A) \sqcap W'(\neg A)] \rightsquigarrow W'(A) \\
 &= [W'(A) \sqcap (\sim W'(A))] \rightsquigarrow W'(A) \\
 &= [W(A) \sqcap (\sim W(A))] \rightsquigarrow W(A) \\
 &= [t \sqcap (\sim t)] \rightsquigarrow t \\
 &= [t \sqcap f] \rightsquigarrow t \\
 &= f \rightsquigarrow t = t
 \end{aligned}$$

Consider the valuation $W(A) = f$, and the truth valuation W' extending W . Using the same method we can determine that:

$$W'[(A \wedge \neg A) \rightarrow A] = t$$

Then for every truth valuation, proposition (a) is true, hence (a) is a tautology.

(b), (c) and (d) can be solved similarly.

1.14.6

Prove that the following propositions are contradictions

- (a) $A \wedge \neg A$ (b) $(\neg A \vee (B \wedge \neg B)) \leftrightarrow A$
 (c) $(A \rightarrow B) \wedge (B \rightarrow C) \wedge (A \wedge \neg C)$ (d) $\neg(A \wedge B) \wedge (A \rightarrow B) \wedge A$

Solution:

(d) Let us write all the possible truth valuations of the atoms of (d):

- (i) $W(A) = t$ and $W(B) = t$
- (ii) $W(A) = t$ and $W(B) = f$
- (iii) $W(A) = f$ and $W(B) = t$
- (iv) $W(A) = f$ and $W(B) = f$

We then calculate the corresponding truth valuations W' extending W for each of the above valuations.

$$\begin{aligned}
 \text{(i)} \quad & W'[\neg(A \wedge B) \wedge ((A \rightarrow B) \wedge A)] \\
 &= W'[\neg(A \wedge B)] \sqcap W'((A \rightarrow B) \wedge A) \\
 &= \sim [W'(A \wedge B)] \sqcap [W'(A \rightarrow B)] \sqcap W'(A) \\
 &= \sim [W'(A) \sqcap W'(B)] \sqcap [W'(A) \rightsquigarrow W'(B)] \sqcap W'(A) \\
 &= \sim [W(A) \sqcap W(B)] \sqcap [W(A) \rightsquigarrow W(B)] \sqcap W(A) \\
 &= \sim [t \sqcap t] \sqcap [t \rightsquigarrow t] \sqcap t \\
 &= (\sim t) \sqcap t \sqcap t = f \sqcap t = f.
 \end{aligned}$$

We also calculate the corresponding truth valuations W' for (ii), (iii) and (iv). We have $W'(A \wedge B) \wedge ((A \rightarrow B) \wedge A) = f$ in every case. Hence (d) is an antilogy.

(a), (b) and (c) can be solved similarly.

1.14.7

Complete the following truth table:

A	B	$\neg A$	$\neg B$	$A \rightarrow B$	$\neg A \vee B$	$(A \rightarrow B) \leftrightarrow (\neg A \vee B)$
t	t					
t	f					
f	t					
f	f					

1.14.8

Prove that the following propositions are logically equivalent:

- (a) $\neg(A \wedge B)$ and $\neg A \vee \neg B$
- (b) $A \vee (B \wedge C)$ and $(A \vee B) \wedge (A \vee C)$
- (c) $A \wedge B$ and $B \wedge A$
- (d) $A \rightarrow B$ and $\neg B \rightarrow \neg A$

Solution:

- (b) Let us construct the truth table of the two propositions and check the similarity of the last two columns.

A	B	C	$B \wedge C$	$A \vee B$	$A \vee C$	$A \vee (B \wedge C)$	$(A \vee B) \wedge (A \vee C)$
t	t	t	t	t	t	t	t
t	t	f	f	t	t	t	t
t	f	t	f	t	t	t	t
t	f	f	f	t	t	t	t
f	t	t	t	t	t	t	t
f	t	f	f	t	f	f	f
f	f	t	f	f	t	f	f
f	f	f	f	f	f	f	f

Thus $A \vee (B \wedge C)$ and $(A \vee B) \wedge (A \vee C)$ are logically equivalent.

- (a), (c) and (d) can be solved similarly.

1.14.9

Prove by truth tables that the following propositions are tautologies:

- (a) $(A \vee B) \vee C \leftrightarrow A \vee (B \vee C)$
- (b) $(A \wedge B) \wedge C \leftrightarrow A \wedge (B \wedge C)$
- (c) $A \vee B \leftrightarrow B \vee A$

- (d) $A \leftrightarrow \neg\neg A$
 (e) $A \wedge (B \vee \neg B) \leftrightarrow A$
 (f) $A \vee (B \wedge \neg B) \leftrightarrow A$
 (g) $A \wedge (B \vee C) \leftrightarrow (A \wedge B) \vee (A \wedge C)$
 (h) $A \vee (B \wedge C) \leftrightarrow (A \vee B) \wedge (A \vee C)$
 (i) $\neg(A \wedge B) \leftrightarrow \neg A \vee \neg B$
 (j) $\neg(A \vee B) \leftrightarrow \neg A \wedge \neg B$

1.14.10

If $S = \{A \vee B, A \rightarrow C\}$, prove that $S \models B \vee C$.

Solution:

We construct the truth table of the propositions of S :

A	B	C	$A \wedge B$	$A \rightarrow C$	
t	t	t	t	t	(1)
t	t	f	f	t	
t	f	t	f	t	(2)
t	f	f	f	t	
f	t	t	t	t	(3)
f	t	f	f	t	(4)
f	f	t	f	f	
f	f	f	f	f	

We observe that there are four valuations validating all the propositions of S . Let us determine whether each one of the truth valuations also validates $B \vee C$:

$$\begin{aligned}
 (1) \quad W(A) = t, \quad W(B) = t, \quad W(C) = t \\
 W(B \vee C) = W(B) \sqcup W(C) = t \sqcup t = t
 \end{aligned}$$

We can observe by the same method that $W(B \vee C) = t$ also holds for (2), (3) and (4). Hence $S \models B \vee C$.

1.14.11

If $S = \{A \leftrightarrow C, B \leftrightarrow D, (A \vee B) \wedge (C \vee D)\}$ prove that $S \not\models (A \wedge B) \vee (C \wedge D)$.

Solution:

As in Exercise 1.14.10, we construct the truth table of the propositions of S , and determine the truth valuations validating the elements of S . There are three such truth valuations. We then calculate the corresponding truth valuations of $(A \wedge B) \vee (C \wedge D)$. One of these valuations does not validate $(A \wedge B) \vee (C \wedge D)$.

1.14.12

Prove that if $\{A, \neg B\} \models C \wedge \neg C$, then $\{A\} \models B$.

Solution:

Let us assume $\{A\} \not\models B$. Then there is a truth valuation W such that $W(A) = t$ and $W(B) = f$, and thus $W(\neg B) = t$. Hence W validates both A and $\neg B$, and therefore validates $C \wedge \neg C$. In other words, $W(C \wedge \neg C) = t$. But then $W(C) \sqcap W(\neg C) = t$, hence $W(C) = W(\neg C) = t$ and $W(\neg C) = \sim W(C) = \sim t = f$, which is a contradiction. Therefore $\{A\} \models B$.

1.14.13

S_1 and S_2 are two sets of PL propositions. Determine which of the following claims is true:

- (a) $Con(S_1 \cup S_2) = Con(S_1) \cup Con(S_2)$
- (b) $Con(S_1 \cap S_2) = Con(S_1) \cap Con(S_2)$

If the corresponding claim is not true, give a counterexample. If it is true, write the corresponding proof.

Solution:

- (a) $Con(S_1) \cup Con(S_2) \subseteq Con(S_1 \cup S_2)$:

Consider $\sigma \in Con(S_1) \cup Con(S_2)$. Then either $\sigma \in Con(S_1)$ or $\sigma \in Con(S_2)$. Since $S_1 \subseteq S_1 \cup S_2$ we have $Con(S_1) \subseteq Con(S_1 \cup S_2)$ (Corollary 1.5.8), and since $S_2 \subseteq S_1 \cup S_2$ we have $Con(S_2) \subseteq Con(S_1 \cup S_2)$.

Then $\sigma \in \text{Con}(S_1 \cup S_2)$. In other words:

$$\text{Con}(S_1) \cup \text{Con}(S_2) \subseteq \text{Con}(S_1 \cup S_2)$$

However, $\text{Con}(S_1 \cup S_2) \subseteq \text{Con}(S_1) \cup \text{Con}(S_2)$ does not hold true:

Consider $S_1 = \{A\}$, $S_2 = \{A \rightarrow B\}$. Then for all valuations W validating both A and $A \rightarrow B$, we have by definition that W also validates B . Then

$$B \in \text{Con}(S_1 \cup S_2) \quad (1)$$

However, $B \notin \text{Con}(S_1)$; for example, consider a valuation W_1 such that $W_1(A) = t$ and $W_1(B) = f$. Moreover, $B \notin \text{Con}(S_2)$; for example, consider a valuation W_2 such that $W_2(A \rightarrow B) = t$, while $W_2(A) = f$ and $W_2(B) = t$.

Then $B \notin \text{Con}(S_1) \cup \text{Con}(S_2)$, and by (1)

$$\text{Con}(S_1 \cup S_2) \not\subseteq \text{Con}(S_1) \cup \text{Con}(S_2)$$

Hence claim (a) is not true.

(b) We will give a counterexample:

Let us assume that $S_1 \cap S_2 = \emptyset$. By corollary 1.5.4, we have $\text{Con}(S_1 \cap S_2) = \text{Con}(\emptyset)$, which is the set of all tautologies. The other way round, let us assume that $S_1 = \{A\}$ and $S_2 = \{B\}$. Then $A \vee B$ belongs to $\text{Con}(S_1)$ as well as to $\text{Con}(S_2)$, since every truth valuation validating all the elements of S_1 or all the elements of S_2 , also validates $A \vee B$. However $A \vee B$ is not a tautology (why?). Hence claim (b) is not true.

1.14.14

Prove that if $S \cup \{A\} \models B$, then $S \models A \rightarrow B$.

Solution:

$S \cup \{A\} \models B$. Then every truth valuation W validating all the propositions of $S \cup \{A\}$ also validates B . This is to say that, for every truth valuation W for which $W(C) = t$ holds for every $C \in S$, we have $W(A) = t$ and $W(B) = t$. Then $W(A \rightarrow B) = W(A) \rightsquigarrow W(B) = t \rightsquigarrow t = t$. Hence $S \models A \rightarrow B$.

1.14.15

Given that S is a consistent set of propositions, prove that

$$S \cup \{\varphi\} \text{ inconsistent} \Leftrightarrow S \models \neg\varphi$$

Solution:

$$(i) \quad S \cup \{\varphi\} \text{ inconsistent} \Rightarrow S \models \neg\varphi.$$

Let us assume $S \cup \{\varphi\}$ inconsistent. Since S is consistent, there is at least one truth valuation W such that, for every $\sigma \in S$, $W(\sigma) = t$ holds. If $W(\varphi) = t$, then $S \cup \{\varphi\}$ is consistent, which contradicts our assumption. Hence $W(\varphi) = f$; which means that $W(\neg\varphi) = t$ for every truth valuation W such that, for every $\sigma \in S$, $W(\sigma) = t$ holds. Hence $S \models \neg\varphi$.

$$(ii) \quad S \models \neg\varphi \Rightarrow S \cup \{\varphi\} \text{ is inconsistent.}$$

Let us assume $S \models \neg\varphi$. Then for every truth valuation W such that, for every $\sigma \in S$, $W(\sigma) = t$ holds, we also have $W(\neg\varphi) = t$, and therefore $W(\varphi) = f$. This is to say that, if the elements of S take value t , then φ takes value f . But then $S \cup \{\varphi\}$ is inconsistent.

1.14.16

Write in PL the propositions of the following sets of propositions and determine which of the sets are consistent:

S_1 : The witness was scared or, if John committed suicide, a note was found.
If the witness was scared, then John killed himself.

S_2 : Love is blind and happiness is at reach, or love is blind, and women are more intelligent than men. If happiness is at reach, then love is not blind.
Women are not more intelligent than men.

Solution:

Formalise the propositions occurring in S_1 and S_2 and construct the truth tables of S_1 and S_2 . Use Definition 1.5.5.

1.14.17

If $S_1 \subseteq \text{Con}(S_2)$ then $\text{Con}(S_1 \cup S_2) = \text{Con}(S_2)$.

1.14.18

Let Q be a set of propositions and S a consistent subset of Q . Then S is said to be a **maximal consistent set** if, for every $S' \subseteq Q$ with $S \subset S'$, S' is inconsistent.

Prove that a consistent set S is a maximal consistent set if and only if, for every proposition $\sigma \in Q$, only one of the following two propositions holds:

$$(i) \quad \sigma \in S \qquad (ii) \quad \neg\sigma \in S$$

Solution:

(\Rightarrow) Let us assume that S is a maximal consistent set. Consider $\sigma \in Q$ such that $\sigma \notin S$.

Then S is a proper subset of $S' = S \cup \{\sigma\}$, which is inconsistent by definition of S . So if a truth valuation W validates all the propositions of S , it must refute σ . Since $W(\sigma) = f$, we have $W(\neg\sigma) = t$. However, $\neg\sigma$ belongs to S ; otherwise S would be a proper subset of $S'' = S \cup \{\neg\sigma\}$, and we would have $W(\varphi) = t$ for every $\varphi \in S$, as well as $W(\neg\sigma) = t$. In other words, S'' would be consistent, which contradicts S being maximal consistent.

In the same way, we can prove that if $\neg\sigma \notin S$ then $\sigma \in S$.

(\Leftarrow) Let us assume that S is consistent, and that for every $\sigma \in Q$, only one of $(\sigma \in S)$ and $(\neg\sigma \in S)$ holds. We will prove that S is a maximal consistent set.

Let S' be a set such that $S \subset S'$. Consider $\varphi \in (S' - S)$. We then have $\varphi \in S'$, $\varphi \notin S$, and $\neg\varphi \in S$ (why?). However, S is consistent. Hence, for every truth valuation W validating all the propositions of S , we have $W(\neg\varphi) = t$, or $W(\varphi) = f$. But in that case, there is no truth valuation validating the propositions of S' , since $\neg\varphi \in S \subset S'$ and $\varphi \in S'$. Hence, since every S' such that $S \subset S'$ is inconsistent, S is maximal consistent.

The meaning of o_i , $1 \leq i \leq 16$, will appear more clearly in:

$A \ o_1 \ B$	$\equiv A \vee \neg A$	(truth)
$A \ o_2 \ B$	$\equiv \neg(A \wedge B)$	($A \mid B$ in Example 1.6.5)
$A \ o_3 \ B$	$\equiv A \rightarrow B$	
$A \ o_4 \ B$	$\equiv \neg A$	
$A \ o_5 \ B$	$\equiv B \rightarrow A$	
$A \ o_6 \ B$	$\equiv \neg B$	
$A \ o_7 \ B$	$\equiv A \leftrightarrow B$	
$A \ o_8 \ B$	$\equiv \neg(A \vee B)$	($A : B$ in Example 1.6.5)
$A \ o_9 \ B$	$\equiv A \vee B$	
$A \ o_{10} \ B$	$\equiv \neg(A \leftrightarrow B)$	
$A \ o_{11} \ B$	$\equiv B$	
$A \ o_{12} \ B$	$\equiv \neg(A \rightarrow B)$	
$A \ o_{13} \ B$	$\equiv A$	
$A \ o_{14} \ B$	$\equiv \neg(B \rightarrow A)$	
$A \ o_{15} \ B$	$\equiv A \wedge B$	
$A \ o_{16} \ B$	$\equiv A \wedge \neg A$	(contradiction)

The above truth table also contains logical connectives which refer to only one of the PL propositions, such as in o_4 and o_6 . The exercise thus provides us with a more general conclusion involving all the PL connectives.

For every $o \in \{o_1, \dots, o_{16}\}$ we have: If $A \ o \ B$ takes value t when A and B take value t , then $\{o\}$ is not a sufficient set, since the negation \neg cannot be expressed by means of o : whatever the number of occurrences of o in the proposition expressing the negation, we will always have truth value t and never f , namely $\sim t$, the value of $\neg A$, $\neg B$. Thus $o \neq o_1, o_3, o_5, o_7, o_9, o_{11}, o_{13}, o_{15}$.

If $A \ o \ B$ takes value f when A and B take value f , then $\{o\}$ is not sufficient, since it can only provide truth value f and never t , which is the value of $\neg A$, $\neg B$. Thus $o \neq o_9, o_{10}, o_{11}, o_{12}, o_{13}, o_{14}, o_{15}, o_{16}$.

If $o = o_4$, then $\{o\}$ is not sufficient since $\neg\neg A \equiv A$. We will always have A or $\neg A$, no matter how many times the negation recurs, and we will thus not be able to express, for instance, \wedge and \vee . Hence $o \neq o_4$, and for the same reason, $o \neq o_6$.

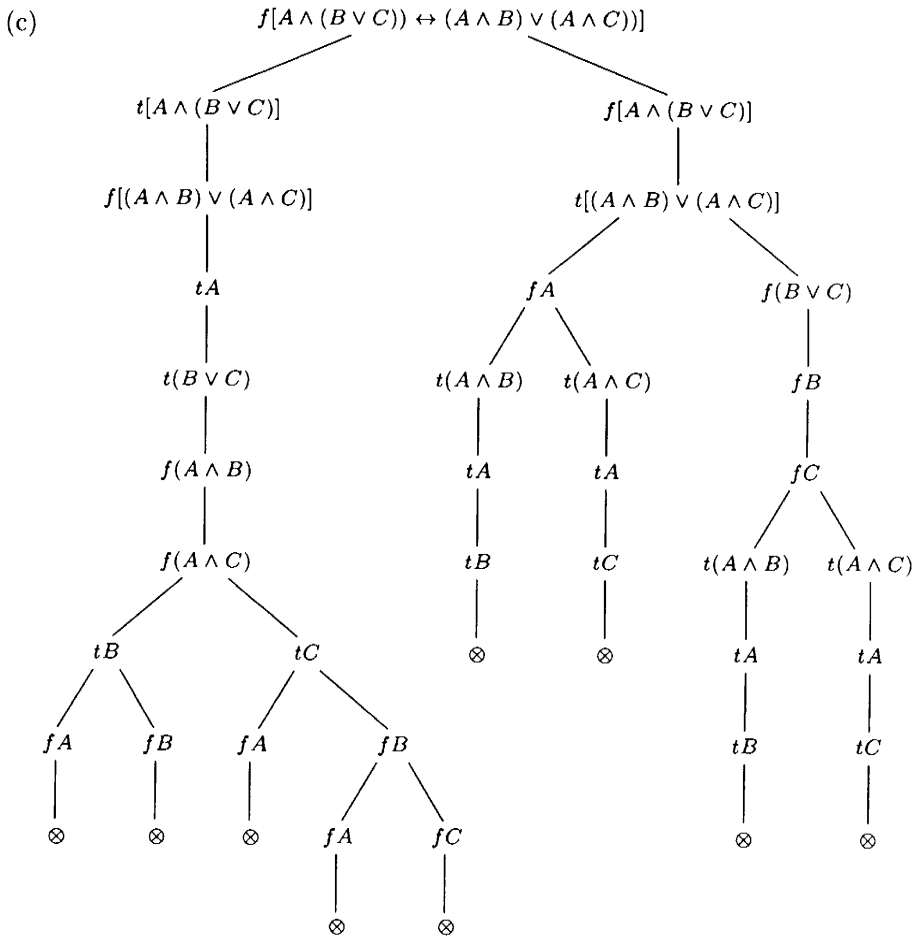
If $o = o_2$ or $o = o_8$, then $\{o\}$ is sufficient, as we saw in Example 1.6.5.

1.14.21

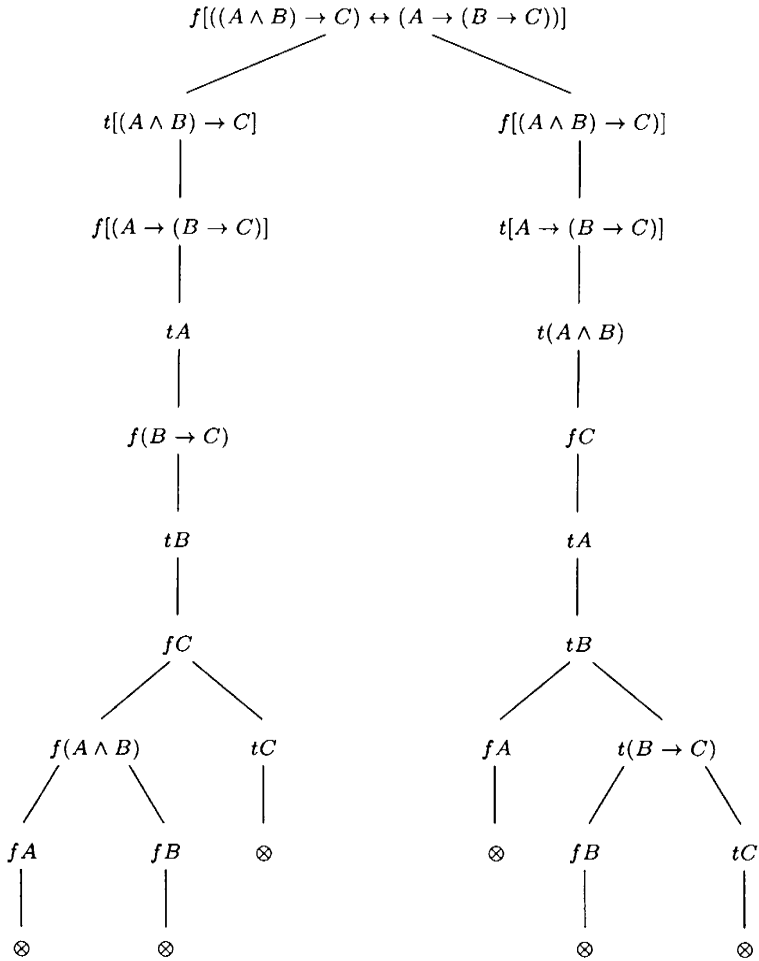
Prove the following propositions by using the semantic tableaux method:

- (a) $\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$
- (b) $(A \wedge (A \rightarrow B)) \rightarrow B$
- (c) $(A \wedge (B \vee C)) \leftrightarrow ((A \wedge B) \vee (A \wedge C))$
- (d) $((A \rightarrow B) \wedge (A \rightarrow C)) \rightarrow (A \rightarrow (B \wedge C))$
- (e) $((B \rightarrow A) \wedge (C \rightarrow A)) \rightarrow ((B \vee C) \rightarrow A)$
- (f) $((A \wedge B) \rightarrow C) \leftrightarrow (A \rightarrow (B \rightarrow C))$
- (g) $(A \rightarrow (B \rightarrow C)) \leftrightarrow (B \rightarrow (C \rightarrow A))$

Solution:



(f)

**1.14.22**

Prove Theorem 1.8.4 of the substitution of equivalences.

Solution:

Hint: first examine the following cases:

σ does not occur in φ .

σ occurs in φ , and σ' and φ are identical.

σ occurs in φ , and σ' and φ are not identical, but φ' and φ are.

Next assume that σ differs from σ' , σ occurs in φ , and φ differs from φ' .

- (1) The number of logical connectives occurring in φ is 0. Then φ is an atomic proposition and φ and σ must be identical. φ' is thus φ or σ' . Then we have $\vdash \varphi \leftrightarrow \varphi'$.
- (2) Let us assume the theorem holds true if φ has n logical connectives.
- (3) We need to prove that if (2) holds true, then the theorem holds if φ has $n + 1$ connectives.

The following cases must be considered separately:

- (i) φ is the proposition $\neg\varphi_1$. We know that $\vdash \varphi_1 \leftrightarrow \varphi'_1$, where φ'_1 is derived from φ_1 by substituting none, one, or more than one occurrence of σ with σ' . By the completeness theorem, we know that $(C \leftrightarrow D) \rightarrow (\neg C \leftrightarrow \neg D)$ is a tautology derivable in our axiomatic system.

$$\vdash \neg\varphi_1 \leftrightarrow \neg\varphi'_1, \varphi \leftrightarrow \varphi'$$

- (ii) φ is the proposition $\varphi_1 \circ \varphi_2$, where $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$. Then $\vdash \varphi_1 \leftrightarrow \varphi'_1$ and $\vdash \varphi_2 \leftrightarrow \varphi'_2$, and by the tautology $C \rightarrow (D \rightarrow (C \wedge D))$, completeness, and the theorem of substitution of equivalences we obtain

$$\vdash (\varphi_1 \leftrightarrow \varphi'_1) \wedge (\varphi_2 \leftrightarrow \varphi'_2)$$

The result follows by the tautology

$$(C_1 \leftrightarrow D_1) \wedge (C_2 \leftrightarrow D_2) \rightarrow (C_1 \circ C_2 \leftrightarrow D_1 \circ D_2)$$

1.14.23

Assuming that $\vdash (A \wedge B) \leftrightarrow \neg(A \rightarrow \neg B)$ and $\neg(A \vee B) \leftrightarrow (\neg A \rightarrow B)$, derive:

- (a) $\vdash (\neg B \rightarrow \neg A) \leftrightarrow (A \rightarrow B)$
- (b) $\vdash (\neg A \rightarrow B) \leftrightarrow (\neg B \rightarrow A)$ and $\vdash (A \rightarrow B) \leftrightarrow (B \rightarrow \neg A)$
- (c) $\vdash (A \wedge B) \rightarrow A$
- (d) $\vdash (A \wedge \neg A) \rightarrow B$
- (e) $\vdash A \vee \neg A$

Solution:

(a) and (b). We use Axiom (3), the Law of Negation, and the Theorem of substitution of equivalences, to prove \rightarrow and then \leftarrow .

(c) $\vdash \neg A \rightarrow (B \rightarrow \neg A)$ (1) by Axiom (1).

$\vdash \neg A \rightarrow (A \rightarrow \neg B)$ (2) by (1), (b), and substitution of equivalences.

$\vdash (A \rightarrow \neg B) \rightarrow A$ (3) by (2), (b), and substitution of equivalences.

But according to the assumptions, this is what we are seeking, namely $(A \wedge B) \rightarrow A$.

We usually need to proceed analytically in order to determine the steps of the derivation: we first try to determine (3), then determine (2) from (3), and continue until a proposition known to be derivable, such as (1), is found.

1.14.24

Using resolution prove that the following sets are not verifiable:

(a) $S = \{\neg A \vee B \vee D, \neg B \vee D \vee A, \neg D \vee C, \neg D \vee A, A \vee B, B \vee \neg C, \neg A \vee \neg B\}$

(b) $S = \{\neg A \vee B, \neg B \vee C, \neg C \vee A, A \vee C, \neg A \vee \neg C\}$

(c) $S = \{A \vee B \vee C, A \vee B \vee \neg C, A \vee \neg B, \neg A \vee \neg C, \neg A \vee C\}$

Solution:

- | | | | |
|--------------|-----|------------------------|---------------------------------|
| (c) Consider | (1) | $A \vee B \vee C$ | |
| | (2) | $A \vee B \vee \neg C$ | |
| | (3) | $A \vee \neg B$ | |
| | (4) | $\neg A \vee \neg C$ | |
| | (5) | $\neg A \vee C$ | |
| | (6) | $A \vee B$ | by (1) and (2) with resolution. |
| | (7) | A | by (3) and (6) with resolution. |
| | (8) | $\neg A$ | by (4) and (5). |
| | (9) | \square | by (7) and (8). |

Hence S is not verifiable.

1.14.25

Using resolution, prove that:

- (a) $\{B \rightarrow A, C \rightarrow C, D \rightarrow B, B \vee C \vee D\} \vdash A \wedge B$
- (b) $\{A \wedge B \rightarrow C, A \rightarrow B\} \vdash A \rightarrow C$.

Solution:

- (b) We just need to prove that $\{A \wedge B \rightarrow C, A \rightarrow B, \neg(A \rightarrow C)\}$ is not verifiable.

Let us convert the propositions of $\{A \wedge B \rightarrow C, A \rightarrow B, \neg(A \rightarrow C)\}$ into a CNF:

$$\begin{aligned}
 A \wedge B \rightarrow C &\leftrightarrow \neg\neg(A \wedge B) \leftrightarrow \neg[\neg(\neg A \vee \neg B)] \vee C \\
 &\leftrightarrow \neg A \vee \neg B \vee C \quad A \rightarrow B \leftrightarrow \neg A \vee B \quad \neg(A \rightarrow C) \\
 &\leftrightarrow \neg(\neg A \vee C) \leftrightarrow A \wedge \neg C \\
 &\leftrightarrow (A \wedge \neg C) \vee (A \wedge \neg A) \leftrightarrow A \wedge (\neg C \vee \neg A)
 \end{aligned}$$

We apply resolution on $\{\neg A \vee \neg B \vee C, \neg A \vee B, A, A \wedge \neg C \vee \neg A\}$:

- (1) $\neg A \vee \neg B \vee C$
- (2) $\neg A \vee B$
- (3) A
- (4) $\neg C \vee \neg A$
- (5) $\neg A \vee \neg B$ by (1) and (4).
- (6) B by (2) and (3).
- (7) $\neg B$ by (3) and (5).
- (8) \square by (6) and (7).

1.14.26

Express the following propositions as sets of clauses:

- (a) $\neg(A \wedge B \wedge \neg C)$
- (b) $A \leftrightarrow (\neg B \wedge \neg C)$

Solution:

- (a) We first need to determine the CNF of $\neg(A \wedge B \wedge \neg C)$:

$$[\neg(A \wedge B \wedge \neg C)] \leftrightarrow [\neg A \vee \neg B \vee \neg \neg C] \leftrightarrow [\neg A \vee \neg B \vee C]$$

Set theoretically: $\{\{\neg A, \neg B, C\}\}$.

1.14.27

Which of the following sets of clauses is satisfiable and why? For every satisfiable set, determine a truth valuation which satisfies it.

- (a) $\{\{A, B\}, \{\neg A, \neg B\}, \{\neg A, B\}\}$ (b) $\{\{\neg A\}, \{A, \neg B\}, \{B\}\}$
 (c) $\{\{A\}, \square\}$ (d) $\{\square\}$

Solution:

- (b) The corresponding CNF is $\neg A \wedge (A \vee \neg B) \wedge B$. This proposition is not satisfiable. If it had been satisfiable, there would be a truth valuation W such that $W(B) = t$ (in order to satisfy $\{B\}$) and $W(A) = f$ (in order to satisfy $\{\neg A\}$). But then W would not satisfy $\{A, \neg B\}$.
 (d) \square is not satisfiable by its definition.

1.14.28

Determine the resolvent $R(S)$ of the set S in the following cases:

- (a) $S = \{\{A, \neg B\}, \{A, B\}, \{\neg A\}\}$
 (b) $S = \{\{A\}, \{B\}, \{A, B\}\}$.

Solution:

- (a) $S = \{\underbrace{\{A, \neg B\}}_1, \underbrace{\{A, B\}}_2, \underbrace{\{\neg A\}}_3\}$, then $R(S) = S \cup \{\underbrace{\{A\}}_{1,2}, \underbrace{\{\neg B\}}_{1,3}, \underbrace{\{B\}}_{2,3}\}$.
 (b) $R(S) = S$. There are no resolvents.

1.14.29

Given the following proposition $Q : (A \wedge B \rightarrow C) \wedge A \wedge (\neg B \rightarrow C)$

- (a) Determine the corresponding set-theoretical form of Q .
- (b) Prove by resolution that $Q \vdash_r C$.
- (c) Given the propositions Q and $A \rightarrow \neg C$, what can we conclude about the truth value of B ?

Solution:

$$\begin{aligned} \text{(a)} \quad [A \wedge B \rightarrow C] &\leftrightarrow [\neg A(\wedge B) \vee C] \leftrightarrow [\neg A \vee \neg B \vee C] \\ [\neg B \rightarrow C] &\leftrightarrow [\neg \neg B \vee C] \leftrightarrow [B \vee C] \end{aligned}$$

$$\text{Hence } Q = \{\{\neg A, \neg B, C\}, \{A\}, \{B, C\}\}$$

- (b) (1) $\{\neg A, \neg B, C\}$
 (2) $\{A\}$
 (3) $\{B, C\}$ by (1) and (2).
 (4) $\{\neg B, C\}$
 (5) $\{C\}$ by (3) and (4).

- (c) $Q' = \{\{\neg A, \neg B, C\}, \{A\}, \{B, C\}, \{\neg A, \neg C\}, \{\neg B\}\}$ is a non-satisfiable set (why?). Thus, every truth valuation validating the propositions of Q and $A \rightarrow \neg C$ assigns truth value f to $\neg B$. In other words, it verifies B .

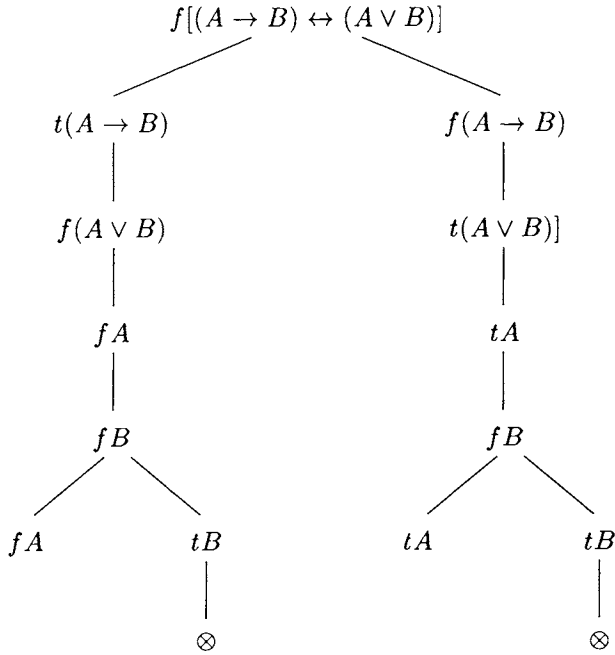
1.14.30

Find truth valuations falsifying the following propositions.

- (a) $(A \rightarrow B) \leftrightarrow (A \vee B)$
- (b) $(A \vee \neg B) \leftrightarrow (A \wedge B)$

Solution:

- (a) Every non-contradictory branch gives a truth valuation W which agrees with the origin of the semantic tableau, Lemma 1.10.5. Hence for every W such that $W(A) = f$ and $W(B) = f$ we have:



$$\begin{aligned}
 W[(A \rightarrow B) \leftrightarrow (A \vee B)] &= W(A \rightarrow B) \leftrightarrow W(A \vee B) \\
 &= [W(A) \rightsquigarrow W(B)] \leftrightarrow [W(A) \sqcup W(B)] \\
 &= [f \rightsquigarrow f] \leftrightarrow [f \sqcup f] \\
 &= t \leftrightarrow f = f
 \end{aligned}$$

1.14.31

If A and $A \rightarrow B$ are Beth-provable then B is also Beth-provable.

Solution:

A is Beth-provable and thus logically true. Then for every truth valuation W we have $W(A) = t$. Then $A \rightarrow B$ is logically true and therefore, for every truth valuation W , we have $W(A \rightarrow B) = t$. Then

$$t = W(A \rightarrow B) = W(A) \rightsquigarrow W(B) = t \rightsquigarrow W(B)$$

But then $W(B) = t$ for every truth valuation W . Hence B is Beth-provable.

1.14.32

- Prove: (a) $\{A \vee B, A \rightarrow C, B \rightarrow D\} \vdash C \vee D$
 (b) $\{A \rightarrow (B \rightarrow C), \neg D \vee A, B\} \vdash D \rightarrow C$

Solution: (a)

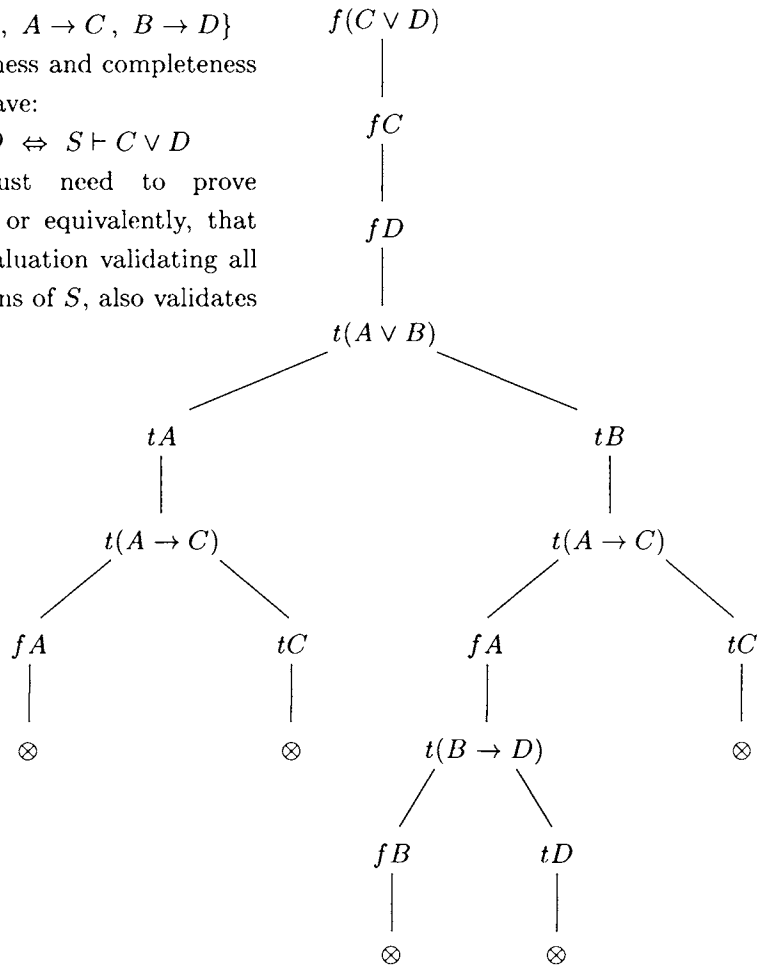
Consider:

$$S = \{A \vee B, A \rightarrow C, B \rightarrow D\}$$

By the soundness and completeness theorem we have:

$$\models C \vee D \Leftrightarrow S \vdash C \vee D$$

Hence we just need to prove $S \models C \vee D$, or equivalently, that every truth valuation validating all the propositions of S , also validates $C \vee D$.



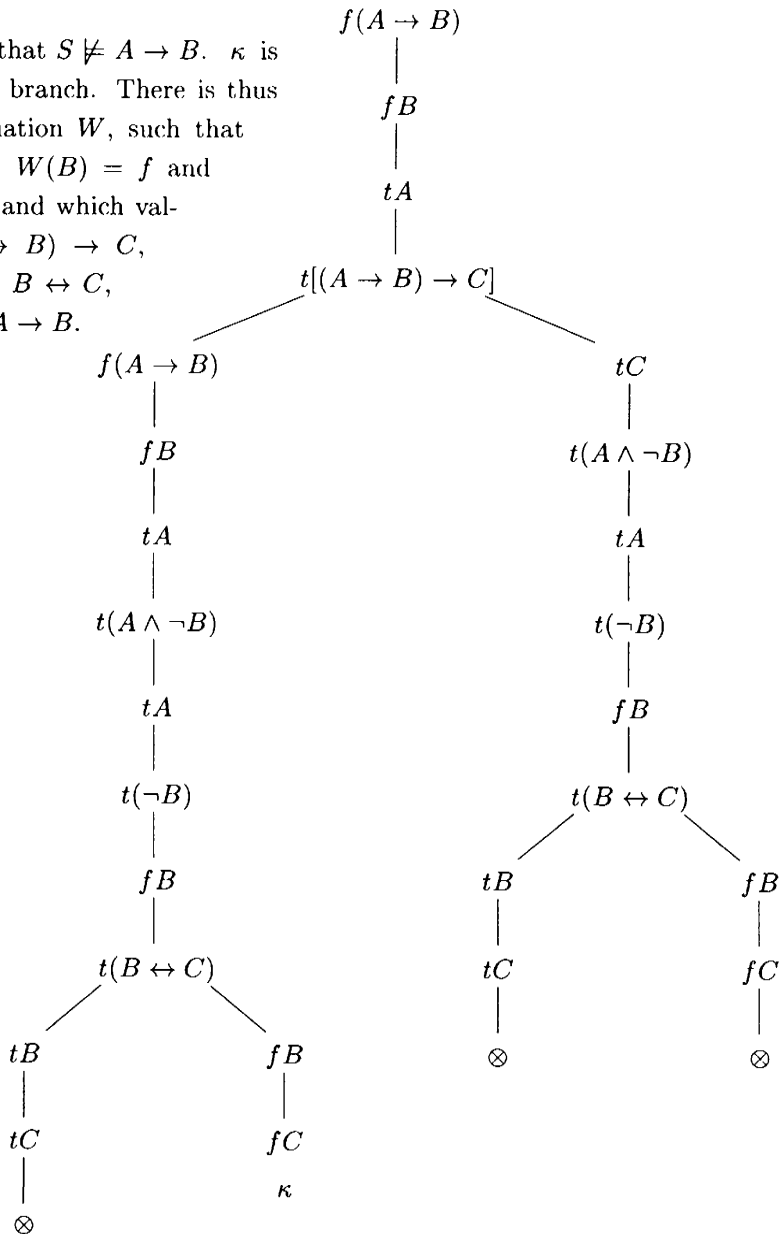
We thus construct a semantic tableau with $f(C \vee D)$ at the origin by concatenating successively the nodes $t(A \vee B)$, $t(A \rightarrow C)$ and $t(B \rightarrow D)$. If $C \vee D$ is verified by all truth valuations which validate the propositions of S , we will have constructed a contradictory semantic tableau.

1.14.33

- Determine whether: (a) $\{(A \rightarrow B) \rightarrow C, A \wedge \neg B, B \leftrightarrow C\} \vdash A \rightarrow B$.
 (b) $\{A \rightarrow B, C \vee B, D \rightarrow (A \vee C), D\} \vdash B$.

Solution: (a)

We observe that $S \not\models A \rightarrow B$. κ is not a closed branch. There is thus a truth valuation W , such that $W(A) = t$, $W(B) = f$ and $W(C) = f$; and which validates $(A \rightarrow B) \rightarrow C$, $A \wedge \neg B$ and $B \leftrightarrow C$, but refutes $A \rightarrow B$.



1.14.34

Prove that if $S \cup \{A\} \vdash B$ then $S \vdash A \rightarrow B$.

Solution:

From the soundness and completeness theorem, we have:

$$S \cup \{A\} \vdash B \Leftrightarrow S \cup \{A\} \models B.$$

Let us assume $S \cup \{A\} \models B$. Then for every truth valuation W validating the propositions of S , we have $W(B) = t$ if $W(A \rightarrow B) = W(A) \rightsquigarrow W(B) = t$, since $W(A \rightarrow B)$ can take truth value f only if $W(B) = f$ and $W(A) = t$. However, then $S \models A \rightarrow B$. Hence $S \cup \{A\} \models B \Rightarrow S \models A \rightarrow B$.

By the completeness theorem we have $S \models A \rightarrow B \Rightarrow S \vdash A \rightarrow B$.

1.14.35

Let $\sigma_1, \sigma_2, \dots$ be an infinite sequence of propositions. If for every i , $\sigma_{i+1} \rightarrow \sigma_i$ is Beth-provable, while $\sigma_i \rightarrow \sigma_{i+1}$ is not, prove that there is no proposition τ such that (a) and (b) both hold true:

(a) For every i , $\tau \rightarrow \sigma_i$ is Beth-provable

(b) $\{\sigma_1, \sigma_2, \dots\} \models \tau$.

Solution:

- (a) We will prove that if for every i , $\tau \rightarrow \sigma_i$ is Beth-provable, then we have: $\{\sigma_1, \sigma_2, \dots\} \models \tau$, or equivalently, $\{\neg\tau, \sigma_1, \sigma_2, \dots\}$ is consistent; in other words, there is a truth valuation W such that:

$$W(\neg\tau) = W(\sigma_1) = W(\sigma_2) = \dots = t$$

Since $\sigma_i \rightarrow \sigma_{i+1}$ is not Beth-provable for every i , there is a truth valuation W such that, for every $n \in \mathbb{N}$, we have:

$$W(\sigma_n \rightarrow \sigma_{n+1}) = f \quad \text{or} \quad W(\sigma_n) \rightsquigarrow W(\sigma_{n+1}) = f.$$

Then

$$W(\sigma_n) = t \quad \text{and} \quad W(\sigma_{n+1}) = f \quad (1)$$

Since $\sigma_{i+1} \rightarrow \sigma_i$, for all $i \in \mathbb{N}$, is Beth-provable, then *for all truth valuations* W' and for all $n \in \mathbb{N}$, $W'(\sigma_{n+1} \rightarrow \sigma_n) = t$; which is equivalent to $W'(\sigma_{n+1}) \rightsquigarrow W'(\sigma_n) = t$, hence:

$$W'(\sigma_{n+1}) = f \quad \text{or} \quad W'(\sigma_n) = t$$

And in a similar way:

$$\begin{aligned} W'(\sigma_n) = f & \quad \text{or} \quad W'(\sigma_{n-1}) = t \\ \dots & \quad \dots \quad \dots \\ W'(\sigma_2) = f & \quad \text{or} \quad W'(\sigma_1) = t \end{aligned} \tag{2}$$

For $W' = W$, we have by (1) that $W(\sigma_{n+1}) = f$ and $W(\sigma_n) = t$. Then by (2) we have:

$$W(\sigma_n) = W(\sigma_{n-1}) = \dots = t \tag{3}$$

For every i , $\tau \rightarrow \sigma_i$ is Beth-provable. Then:

$$t = W(\tau \rightarrow \sigma_{n+1}) = W(\tau) \rightsquigarrow W(\sigma_{n+1}) = W(\tau) \rightsquigarrow f$$

i.e., $W(\tau) = f$. Then:

$$W(\neg\tau) = \sim W(\tau) = t \tag{4}$$

Then there exists a truth valuation W such that (1), (3) and (4) hold true. So for all $n \in \mathbb{N}$, the set $\{\neg\tau, \sigma_1, \sigma_2, \dots, \sigma_n\}$ is consistent.

- (b) Suppose $\{\sigma_1, \sigma_2, \dots\} \models \tau$. We will prove that there exists a truth valuation W , and a $k \in \mathbb{N}$, with the property that $W(\tau \rightarrow \sigma_k) = f$, or equivalently, $W(\tau) = t$ and $W(\sigma_k) = f$.

Since $\{\sigma_1, \sigma_2, \dots\} \models \tau$, the set $\{\neg\tau, \sigma_1, \sigma_2, \dots\}$ is inconsistent. Then the sequence $\neg\tau, \sigma_1, \sigma_2, \dots$ is not satisfiable. From the compactness theorem, there is an $n \in \mathbb{N}$, such that the sequence $\neg\tau, \sigma_1, \sigma_2, \dots, \sigma_n$ is not satisfiable, and the set $\{\neg\tau, \sigma_1, \sigma_2, \dots, \sigma_n\}$ is inconsistent.

However, there is a truth valuation W with the property that:

$$W(\sigma_n \rightarrow \sigma_{n+1}) = f$$

Then $W(\sigma_n) \rightsquigarrow W(\sigma_{n+1}) = f$, i.e.,

$$W(\sigma_n) = t \quad \text{and} \quad W(\sigma_{n+1}) = f \quad (5)$$

Since $\sigma_n \rightarrow \sigma_{n-1}$ is Beth-provable,

$$t = W(\sigma_n \rightarrow \sigma_{n-1}) = W(\sigma_n) \rightsquigarrow W(\sigma_{n-1})$$

that is, $W(\sigma_n) = f$ or $W(\sigma_{n-1}) = t$.

From (5) we obtain:

$$W(\sigma_{n-1}) = t \quad (6)$$

In the same way, by the fact that $\sigma_i \rightarrow \sigma_{i-1}$, $1 < i < n - 1$, are Beth-provable, we have:

$$W(\sigma_{n-2}) = \dots = W(\sigma_1) = t \quad (7)$$

By (5), (6) and (7), the truth valuation W validates all the σ_i , $1 < i < n$. But $\{\neg\tau, \sigma_1, \sigma_2, \dots, \sigma_n\}$ is not consistent. Then $W(\neg\tau) = f$, in other words $W(\tau) = t$. Then by (5) we have:

$$W(\tau \rightarrow \sigma_{n+1}) = W(\tau) \rightsquigarrow W(\sigma_{n+1}) = t \rightsquigarrow f = f$$

Hence $\tau \rightarrow \sigma_{n+1}$ is not Beth-provable.

II Predicate Logic

Συλλάψεις ὅλα καὶ οὐχ ὅλα,
συμφερόμενον διαφερόμενον, συνᾶδον
διᾶδον, ἐκ πάντων ἓν καὶ ἐξ ἑνὸς πάντα.

Things taken together are whole and not whole,
something which is being brought together and
brought apart, which is in tune and out of tune:
out of all things can be made a unity, and out
of a unity, all things.

Heraklith

2.1 Introduction

In the first chapter, we gave an analytic description of the PL language, a formal language, by means of which we can express simple as well as compound propositions. Furthermore, we examined methods of derivation of conclusions from sets of PL propositions.

Even though the PL language is quite rich, it only allows a limited formulation of properties and relations. Let us take as an example the following proposition in the English language:

S : “If George is human, then George is mortal”.

If A denotes the proposition

“George is human”

and B denotes

“George is mortal”

then, within the PL context, S becomes:

$$S : A \rightarrow B$$

S expresses certain qualities of a particular person, namely George. The following question arises: how can we express similar properties of other people such as Socrates or Peter for example? One solution would be to introduce as many propositional symbols as there are different people! However this is impossible in practice.

In this chapter we will describe the language of Predicate Logic which provides a solution to such problems. The new element of this language is the introduction of **variables** and **quantifiers**.

The Variables

If we consider the proposition S and if we assume that x is a variable which takes values from the set of the names of all the people, for instance

$$x = \text{George} \quad \text{or} \quad x = \text{John} \quad \text{or} \quad x = \dots$$

and if “Human” and “Mortal” are symbols denoting properties, then we can represent the general relation between these properties by:

$$P : \text{Human}(x) \rightarrow \text{Mortal}(x)$$

Such representations as “Human(x)” or “Mortal(x)” which express general relations as properties are called **predicates**. A **formula** is a representation like P consisting of predicates connected by logical connectives.

The substitution of the variable x by the constant “George” converts P into the formula

$$S' : \text{Human}(\text{George}) \rightarrow \text{Mortal}(\text{George})$$

Furthermore, if the variable x takes the value “Socrates”, the result will consist of a new formula representing the relation between Socrates and the property of being mortal. “John”, “George” and “Peter” are **constants** in our new formal language.

In general, the correspondence between the variables and the constants on the one hand and the symbols of the English language on the other can be intuitively represented by:

English language		Formal language
pronoun	\longmapsto	variable
proper name	\longmapsto	constant

The special symbols “Human” and “Mortal” are called **predicate symbols**. Predicates can refer to more than one variable, thus expressing not only properties but also relations between many objects. For example, if the variables x and y take values from the set of integers and if we introduce the predicate I , “greater”, we can express one of the fundamental relations of the integers:

$$I(X, Y) : \text{greater}(x, y)$$

which is **interpreted** as “ x is greater than y ”.

If, in the above expression, we replace x by 5 and y by 3, we obviously have a particular version of I :

$$I(5, 3) : \text{greater}(5, 3)$$

which holds true for the integers.

The Quantifiers

The introduction of variables results in the change of the validity of a formula. Consider for example the formula

$$Q(x, y) : \text{flight_}XA(x, y)$$

which is **interpreted** as

$$X \text{ Airlines flight connecting the cities } x \text{ and } y$$

The *validity* of this formula is only *partial*, since there may not be for instance, an X Airlines flight from New York to New Delhi. Conversely, the formula

$$P(x) : \text{Human}(x) \rightarrow \text{Mortal}(x)$$

has a *universal validity*, it holds true for every variable x .

In Predicate Logic, PrL for short, the general or partial validity is denoted by two special symbols, the quantifiers; we thus use a **universal quantifier** and an **existential** one, denoted respectively by \forall and \exists . Hence the initial formula P becomes:

$$P(x) : (\forall x) (\text{Human}(x) \rightarrow \text{Mortal}(x))$$

and Q becomes

$$Q(x, y) : (\exists(x, y)) \text{ flight_} X A(x, y)$$

In the following sections we will formally introduce the language of Predicate Logic.

2.2 The Language of Predicate Logic

We will now give the formal description of a language of PrL, [Chur56, Curr63, Dela87, Hami78, Klee52, Mend64, Meta85, Smul68].

Definition 2.2.1: A PrL language consists of the following fundamental symbols:

(I) Logical symbols:

- (i) *Variables* $x, y, z, \dots, x_0, y_0, z_0, \dots, x_i, \dots$
- (ii) *Logical connectives* $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$
- (iii) *Comma, parentheses* $, ()$
- (iv) *Quantifiers* \forall, \exists

(II) Specific symbols:

- (i) *Predicate symbols* $P, Q, R, \dots, P_0, Q_0, R_0, \dots, P_1, \dots$
- (ii) *Symbols of constants* $a, b, \dots a_0, b_0, \dots, a_1, \dots, a_2, \dots$
- (iii) *Symbols of functions* $f, g, f_0, g_0, f_1, \dots$ ■ 2.2.1

The number of different variables occurring in a predicate symbol is the **degree** or **arity** of the predicate. For example, $Q(x, y, z)$ is a predicate of degree 3 or a 3-ary predicate.

Each quantifier is **dual** to the other one: \forall is equivalent to the sequence of symbols $\neg\exists\neg$ and \exists is equivalent to $\neg\forall\neg$. For the formula $(\forall x) Q(x)$ we have for instance:

$$(\forall x) Q(x) \leftrightarrow \neg(\exists x) \neg Q(x)$$

Each language of PrL, i.e., each set of specific symbols contains all logical symbols. Thus, in order to determine a language, a definition of its specific symbols suffices.

Example 2.2.2: $\mathcal{L}_A = (=, \leq, +, *, 0, 1)$ is a language for arithmetic.

$=$ and \leq are 2-ary predicate symbols:

$=(x, y)$ reads “ $x = y$ ”, and $\leq(x, y)$ denotes “ $x \leq y$ ”.

$+$ and $*$ are 3-ary predicates:

$+(x, y, z)$ reads “ $x + y = z$ ”, and $*(x, y, z)$ reads “ $x * y = z$ ”.

0 and 1 are symbols of constants.

■ 2.2.2

Definition 2.2.3: A **term** is inductively defined by:

- (i) A **constant** is a term.
- (ii) A **variable** is a term.
- (iii) If f is a n -ary **function** and t_1, \dots, t_n are terms then $f(t_1, \dots, t_n)$ is a term.

■ 2.2.3

Definition 2.2.4: An **atomic formula** or **atom** is every sequence of symbols $P(t_1, \dots, t_n)$ where P is an n -ary predicate symbol and t_i is a term, for every $i = 1, 2, \dots, n$.

■ 2.2.4

Definition 2.2.5: A **formula** is inductively defined by:

- (i) Every **atom** is a formula.
- (ii) If σ_1, σ_2 are formulae then $(\sigma_1 \wedge \sigma_2)$, $(\sigma_1 \vee \sigma_2)$, $(\sigma_1 \rightarrow \sigma_2)$, $(\neg\sigma_1)$ and $(\sigma_1 \leftrightarrow \sigma_2)$ are also formulae.
- (iii) If v is a variable and φ is a formula then $((\exists v)\varphi)$, $((\forall v)\varphi)$ are also formulae.
- (iv) Only the sequences of symbols formed according to (i), (ii) and (iii) are formulae.

■ 2.2.5

Example 2.2.6: The following expressions are formulae:

$$(i) \quad \varphi_1 : (\forall y) (\exists x) [P(x, f(y)) \vee Q(x)]$$

$$(ii) \quad \varphi_2 : (\forall x) (\exists y) [P(x) \vee Q(x, y) \rightarrow \neg(R(x))]. \quad \blacksquare \quad 2.2.6$$

Remark 2.2.7: We observe that the definition of the formulae of the language allows trivial uses of quantifiers such as:

$$(\exists x) [y = 3]$$

which is equivalent to the formula:

$$y = 3$$

The trivial uses are formally accepted, however they are usually utilised only in technical proofs. \blacksquare 2.2.7

Example 2.2.8: Here are several formulae of the language \mathcal{L}_A which was defined in Example 2.2.2.

- | | | |
|-----|--|---------------------|
| (1) | $(\forall x) (x = x)$ | = is reflexive |
| (2) | $(\forall x) (\forall y) (x = y \rightarrow y = x)$ | = is symmetric |
| (3) | $(\forall x) (\forall y) (\forall v) [(x = y \wedge y = v) \rightarrow x = v]$ | = is transitive |
| (4) | $(\forall x) (x \leq x)$ | \leq is reflexive |

\blacksquare 2.2.8

We will continue with some definitions that are necessary for the complete description of the PrL context and Logic Programming.

Definition 2.2.9:

- (i) A subsequence t_1 of symbols of a term t , such that t_1 is a term, is a **subterm** of t .
- (ii) A subsequence φ_1 of symbols of a formula φ , such that φ_1 is a formula, is a **subformula** of φ . \blacksquare 2.2.9

Example 2.2.10:

- (i) If $f(x, y)$ is a term, then x , y and $f(x, y)$ are subterms of $f(x, y)$.
- (ii) $P(x)$, $\neg R(x)$, $R(x)$, $P(x) \vee Q(x, y)$ are subformulae of the formula φ_2 of Example 2.2.6. ■ 2.2.10

Remark 2.2.11: In the PL chapter, we only examined the propositions according to their construction based on simple, atomic propositions and logical connectives. We thus assumed that atomic propositions did not require further analysis. Predicate Logic however deals with a more general concept, the atomic formulae. What we assume here is that the atomic formulae consist of predicates and that every n -ary predicate $P(t_1, \dots, t_n)$ expresses a relation between the terms t_1, \dots, t_n .

■ 2.2.11

Definition 2.2.12: Bound and Free Occurrences of Variables:

- (i) An occurrence of a variable v in a formula φ is said to be **bound** if there is a subformula ψ of φ which contains this occurrence and begins with $(\forall v)$ or $(\exists v)$.
- (ii) An occurrence of a variable v in a formula is said to be **free** if it is not bound. ■ 2.2.12

Definition 2.2.13: Free and Bound Variables:

A variable v occurring in a formula φ is said to be **free** if it has at least one free occurrence in φ . v is said to be **bound** if it is not free. ■ 2.2.13

Example 2.2.14: In Example 2.2.6, the variable x has a free occurrence in the subformula

$$\varphi' : (\exists y) (P(x) \vee Q(x, y) \rightarrow \neg R(x))$$

of φ_2 but it is bound in the formula φ_2 itself. Hence x is free for φ' (since it has at least one free occurrence), but it is bound for φ_2 . ■ 2.2.14

Definition 2.2.15: A term with no variables is called a **ground term**.

■ 2.2.15

Example 2.2.16: If a, b are constants and if f is a function symbol, then $a, b, f(a, b), f(f(a), b), \dots$ are ground terms. ■ 2.2.16

Definition 2.2.17: A **sentence** or **closed formula** is a formula with no free variables. ■ 2.2.17

According to the previous definition, in order to form a closed formula from a given one, we have to bind all its free variables with quantifiers.

Example 2.2.18: From the formula $\varphi(x, y) : (x + y = x * y)$ we can form the closed formula

$$\sigma(x, y) : (\forall x) (\exists y) (x + y = x * y) \quad \blacksquare \quad 2.2.18$$

Another way to form propositions is to substitute free occurrences of variables by constants. In general, we have for the substitution [Fitt90]:

Definition 2.2.19: A **substitution set**, or simply **substitution**, is a set:

$$\theta = \{x_1/t_1, x_2/t_2, \dots, x_n/t_n\}$$

where x_i and t_i , $1 \leq i \leq n$, are correspondingly variables and terms such that if $x_i = x_j$, then $t_i = t_j$, $i \leq j \leq n$.

If φ is an expression (atom, term of formula) then $\varphi\theta$ denotes the expression resulting from the substitution of occurrences of x_1, \dots, x_n in φ by the corresponding terms t_1, \dots, t_n .

The **empty substitution** is denoted by E , in other words $E = \{ \}$.

■ 2.2.19

Example 2.2.20: If we use the substitution $\theta = \{x/2, y/2\}$ on the formula:

$$K : \varphi(x, y)$$

of the previous example, we can form the formula:

$$K\theta : (2 + 2 = 2 * 2) \quad \blacksquare \quad 2.2.20$$

Example 2.2.21: Let us consider the formula:

$$\varphi(x, y, z) : (\exists y) R(x, y) \wedge (\forall z) (\neg Q(x, z))$$

and the ground term $f(a, b)$. If we apply the substitution $\{x/f(a, b)\}$ on $\varphi(x, y, z)$ we form the proposition:

$$\varphi(f(a, b), y, z) : (\exists y) R(f(a, b), y) \wedge (\forall z) (\neg Q(f(a, b), z))$$

■ 2.2.21

The basic operation on substitutions is composition:

Definition 2.2.22: Let

$$\theta = \{u_1/s_1, \dots, u_m/s_m\} \quad \text{and} \quad \psi = \{v_1/t_1, \dots, v_n/t_n\}$$

The **composition** of θ and ψ is the substitution:

$$\begin{aligned} \theta\psi = & \{u_1/s_1\psi, \dots, u_n/s_n\psi, v_1/t_1, \dots, v_n/t_n\} \\ & - (\{u_i/s_i\psi \mid u_i = s_i\psi\} \cup \{v_i/t_i \mid v_i \in \{u_1, \dots, u_m\}\}) \end{aligned}$$

■ 2.2.22

In other words we first apply ψ on the terms s_1, \dots, s_m of θ by replacing the variables u_i , u_i of θ by the terms $s_i\psi$, Definition 2.2.19, and we fill in with the elements of ψ . We omit the terms u_i of θ which take the form $s_i\psi$, and the terms v_i of ψ which contain the u_j variables of θ .

Example 2.2.23:

- (1) Let $\theta = \{x/f(y), y/z\}$ (namely $\{u_i/s_i\}$) and $\psi = \{x/a, y/b, z/y\}$ (namely $\{v_i/t_i\}$) be two substitutions. The composition of θ and ψ yields:

$$\begin{aligned} \theta\psi &= \{x/f(y)\psi, y/z\psi, x/a, y/b, z/y\} \\ &\quad - (\{u_i/s_i\psi \mid v_i = s_i\psi\} \cup \{v_i/t_i \mid v_i \in \{u_i\}\}) \\ &= \{x/f(b), y/y, x/a, y/b, z/y\} - (\{y/y\} \cup \{x/a, y/b\}) \\ &= \{x/f(b), z/y\} \end{aligned}$$

(2) Consider the term $t : w(f(v_1), h(x), f(v_2), v_3)$ and the substitutions

$$\begin{aligned}\theta &= \{v_1/f(g(x)), v_2/h(v_1), v_3/h(v_3)\} \quad \text{and} \\ \psi &= \{x/z, v_1/v_2, v_3/v_1\}\end{aligned}$$

Then

$$\begin{aligned}\theta\psi &= \{v_1/f(g(x))\psi, v_2/h(v_1)\psi, v_3/h(v_3)\psi, x/z, v_1/v_2, v_3/v_1\} \\ &\quad - (\emptyset \cup \{v_1/v_2, v_3/v_1\}) \\ &= \{v_1/f(g(z)), v_2/h(v_2), v_3/h(v_1), x/z, v_1/v_2, v_3/v_1\} \\ &\quad - \{v_1/v_2, v_3/v_1\} \\ &= \{v_1/f(g(z)), v_2/h(v_2), v_3/h(v_1), x/z\}\end{aligned}$$

Consequently

$$t(\theta\psi) : w(f(f(g(z))), h(z), f(h(v_2)), h(v_1))$$

Furthermore, we note that:

$$\begin{aligned}t(\theta\psi) &= w(f(f(g(x))), h(x), f(h(v_1)), h(v_3))\psi \\ &= w(f(f(g(z))), h(z), f(h(v_2)), h(v_3)) \\ &= (t\theta)\psi\end{aligned}$$

■ 2.2.23

We can therefore conclude that the associative property of the composition of substitutions holds for the substitution of Example 2.2.23. In general, we have:

Theorem 2.2.24: *For every substitution θ, ψ, γ and for every closed formula σ we have*

- (i) $\theta E = E\theta = \theta$
- (ii) $(\sigma\theta)\psi = \sigma(\theta\psi)$
- (iii) $(\theta\psi)\gamma = \theta(\psi\gamma)$

■ 2.2.24

Definition 2.2.25: Let φ be a formula with no quantifiers and let θ be a substitution. Then $\varphi\theta$ is the formula resulting from the substitution of every term t occurring in φ by $t\theta$.

Correspondingly, if $S = \{C_1, \dots, C_k\}$ is a set of PrL formulae without quantifiers, then $S\theta = \{C_1\theta, \dots, C_k\theta\}$ results from the substitution of C_i , $1 \leq i \leq k$, by the formulae $C_i\theta$. ■ 2.2.25

Definition 2.2.26: Let S_1 and S_2 be two sets of formulae with no quantifiers. S_1 and S_2 are called **variants** if there are two substitutions θ and ψ such that:

$$S_1 = S_2\theta \quad \text{and} \quad S_2 = S_1\psi \quad \blacksquare \quad 2.2.26$$

Example 2.2.27: $S_1 = P(f(x, y), h(z), b)$ and $S_2 = P(f(y, x), g(u), b)$, where b is a constant, are variants. Indeed, if:

$$\begin{aligned} \theta &= \{x/z, y/x, z/u\} & \text{and} & & \psi &= \{x/y, y/x, u/z\} & \text{then} \\ S_1\theta &= P(f(x, y), h(z), b)\theta = P(f(y, x), h(u), b) = S_2 \\ S_2\psi &= P(f(x, y), h(u), b)\psi = P(f(x, y), h(z), b) = S_1 \end{aligned} \quad \blacksquare \quad 2.2.27$$

Definition 2.2.28: A **renaming substitution** is a substitution of the form:

$$\{v_1/u_1, \dots, v_n/u_n\}$$

where v_i and u_i , $1 \leq i \leq n$, are only variables. ■ 2.2.28

2.3 Axiomatic Foundation of Predicate Logic

In section 1.7 of the first chapter, we axiomatized Propositional Logic by means of an axiomatic system consisting of three axioms and a rule. Predicate Logic can be similarly axiomatized [Dela87, Hami78, Klee52, Mend64, Rasi74, RaSi70]. Let us first give an auxiliary definition:

Definition 2.3.1: A **variable** x is **free for the term** t **in the formula** σ , formally $\text{free}(x, t, \sigma)$, if none of the free variables of t becomes bound after the substitution of x by t in all free occurrences of x in σ . ■ 2.3.1

Example 2.3.2: Assume $\sigma : (\forall y) P(x, y)$. Then x is not free for the term y in σ , since, after the substitution x/y in the free occurrences of x , the variable y of the term y is bound.

Conversely, x is free for the term z in σ , where z is a different variable from y ; since, after the substitution x/z in σ , the variables of z , namely z , are not bound. Furthermore, y is free for y in σ ! (σ does not contain free occurrences of y).

■ 2.3.2

Definition 2.3.3: For all formulae φ, τ, σ of PrL, the axioms of PrL are:

- (1) $\varphi \rightarrow (\tau \rightarrow \varphi)$
- (2) $(\varphi \rightarrow (\tau \rightarrow \sigma)) \rightarrow ((\varphi \rightarrow \tau) \rightarrow (\varphi \rightarrow \sigma))$
- (3) $(\neg\varphi \rightarrow \neg\tau) \rightarrow (\tau \rightarrow \varphi)$
- (4) If $\text{free}(x, t, \varphi)$, then the formula

$$(\forall x)\varphi \rightarrow \varphi(x/t)$$

is an axiom.

- (5) If x is not free in the formula φ , then the formula

$$(\forall x)(\varphi \rightarrow \tau) \rightarrow (\varphi \rightarrow (\forall x)\tau)$$

is an axiom.

Just as in PL, the symbol \vdash denotes formulae derivation in the PrL axiomatic system. This axiom system contains two rules:

- (1) Modus Ponens : $\varphi, \varphi \rightarrow \tau \vdash \tau$
- (2) Generalization : $\varphi \vdash (\forall x)\varphi$

■ 2.3.3

Remark 2.3.4:

- (1) According to the rule of generalization, if φ is a formula derived from the axioms and the rules of PrL, then $(\forall x)\varphi$ is also derived in the axiomatic system.

Assume for instance that the following formula is derived:

$$\text{“Human}(x) \rightarrow \text{Mortal}(x)\text{”}$$

Then the formula:

$$\text{“}(\forall x) (\text{Human}(x) \rightarrow \text{Mortal}(x))\text{”}$$

is also derived.

In other words, we can always obtain the validity of a generalized formula $(\forall x) \varphi$ from the validity of the formula φ . An erroneous application of the rule of generalization is often the cause of many mistakes in common discussions. For example, we often claim that:

$$\text{“all politicians are swindlers”}$$

because we know that politicians a and b are swindlers.

However, this claim is not logically valid: in order to generalize, that is to characterise all the politicians and not only a and b , we must be certain that the following formula is derived in our axiomatic system:

$$\text{“politician}(x) \rightarrow \text{swindler}(x)\text{”}$$

That is (hopefully!) not the case.

- (2) Comparing the PL and the PrL axiomatic systems, we note that the axioms and the rule of PL are contained in the axioms and the rules of PrL. However Propositional Logic deals with propositions whereas Predicate Logic refers to a more compound concept, the PrL formulae.
- (3) The quantifier \exists is not included in the axiomatic system since we have already defined \exists as $\neg\forall\neg$ in section 2.1.
- (4) By (2) and Corollary 1.12.2 we conclude that all the tautologies of PL are derived in PrL by considering PrL formulae instead of PL propositions. For example, for the proposition $A \leftrightarrow \neg\neg A$ which is derivable in PL we have the formula $\varphi \leftrightarrow \neg\neg\varphi$ of PrL which is derivable in the axiomatic system defined in Definition 2.3.3. We can thus apply all the tautologies of PL to formulae of PrL (completeness theorem 1.12.1) and have formulae of PrL which are derivable in our PrL axiomatic system. ■ 2.3.4

The theorem of substitution of equivalences is valid in PL as well as in PrL. Its proof is analogous to the proof of the corresponding theorem of PL.

Theorem 2.3.5: Theorem of Substitution of Equivalences for PrL:

If the formula A_1 is derived from the formula A , after the substitution of the formula B by the formula B_1 in none, one or more than one occurrences of B in A , if also $\{x_1, \dots, x_n\}$ are the free variables of B and B_1 which are also bound variables of A , and if

$$\vdash (\forall x_1) \dots (\forall x_n) (B \leftrightarrow B_1)$$

then $\vdash A \leftrightarrow A_1$.

■ 2.3.5

The formulae of PrL which are derived in the axiomatic system of PrL are the only “legitimate” formulae we can work with within the PrL context. The following theorem provides us with a list of the formulae which are most often used. These formulae express associativity and distributivity of quantifiers on logical connectives. As shown in this Theorem, these properties of the quantifiers are not fully valid.

Theorem 2.3.6: *If φ and σ are formulae of PrL, then the following formulae are derived in PrL:*

$$(\forall x) (\varphi \rightarrow \sigma) \rightarrow ((\forall x) \varphi \rightarrow (\forall x) \sigma)$$

$$((\forall x) \varphi \rightarrow (\forall x) \sigma) \rightarrow (\exists x) (\varphi \rightarrow \sigma)$$

$$((\exists x) \varphi \rightarrow (\exists x) \sigma) \rightarrow (\exists x) (\varphi \rightarrow \sigma)$$

$$(\exists x) (\varphi \leftrightarrow \sigma) \rightarrow ((\forall x) \varphi \rightarrow (\exists x) \sigma)$$

$$((\forall x) \varphi \vee (\forall x) \sigma) \rightarrow (\forall x) (\varphi \vee \sigma)$$

$$(\forall x) (\varphi \vee \sigma) \rightarrow ((\exists x) \varphi \vee (\forall x) \sigma)$$

$$(\exists x) (\varphi \vee \sigma) \leftrightarrow ((\exists x) \varphi \vee (\exists x) \sigma)$$

$$(\exists x) (\varphi \wedge \sigma) \rightarrow ((\exists x) \varphi \wedge (\exists x) \sigma)$$

$$(\forall x) (\varphi \wedge \sigma) \rightarrow ((\forall x) \sigma \wedge (\exists x) \sigma)$$

$$(\forall x)(\varphi \wedge \sigma) \leftrightarrow ((\forall x)\varphi \wedge (\forall x)\sigma)$$

$$(\exists y)(\forall x)\varphi \rightarrow (\forall x)(\exists y)\varphi$$

$$(\forall x)(\forall y)\varphi \leftrightarrow (\forall y)(\forall x)\varphi$$

$$(\exists x)(\exists y)\varphi \leftrightarrow (\exists y)(\exists x)\varphi$$

$$(\forall x)\varphi \leftrightarrow \varphi \quad \text{if there is no free occurrence of } x \text{ in } \varphi$$

$$(\exists x)\varphi \leftrightarrow \varphi \quad \text{if there is no free occurrence of } x \text{ in } \varphi \quad \blacksquare \quad 2.3.6$$

Some of the errors which often occur in formal proofs in all sciences are caused by an erroneous use of distributivity of quantifiers on logical connectives. For example, the formulae:

$$(\forall x)\varphi \vee (\forall x)\sigma \rightarrow (\forall x)(\varphi \vee \sigma)$$

and

$$(\exists x)(\varphi \wedge \sigma) \rightarrow ((\exists x)\varphi \wedge (\exists x)\sigma)$$

taken from the above list are derivable in PrL. But the formulae:

$$((\forall x)\varphi \vee (\forall x)\sigma) \leftrightarrow (\forall x)(\varphi \vee \sigma)$$

and

$$(\exists x)(\varphi \wedge \sigma) \leftrightarrow ((\exists x)\varphi \wedge (\exists x)\sigma)$$

which express full distributivity of the quantifiers \forall and \exists on \vee and \wedge respectively, are not valid:

The formulae:

$$(\forall x)(\varphi \vee \sigma) \rightarrow ((\forall x)\varphi \vee (\forall x)\sigma)$$

and

$$((\exists x)\varphi \wedge (\exists x)\sigma) \rightarrow (\exists x)(\varphi \wedge \sigma)$$

are NOT derivable in the axiomatic system of PrL and are NOT valid.

For example, the formula:

$$(\forall x)[(x = 2x) \vee (x \neq 2x)]$$

is true (see Definition 2.4.2).

However:

$$[(\forall x)(x = 2x)] \vee [(\forall x)(x \neq 2x)]$$

is NOT a true formula. If it were true, then at least one of the formulae:

$$(\forall x)(x = 2x), \quad (\forall x)(x \neq 2x)$$

(Definition 2.5.5) would have to be true. That is not the case: if $x = 1$, $x = 2x$ is not true and if $x = 0$, $x \neq 2x$ is not true. We therefore need to be very careful when using commutativity and distributivity of quantifiers, for it is very easy to be misled to erroneous conclusions.

Remark 2.3.7: If we were to extend the language of PrL with a particular logical symbol of arity 2, the equality symbol “=”, there would be properties of “=” which could not be expressed by means of the axiom system of Definition 2.3.3; this axiomatization expresses general properties of a predicate, and is not able to describe properties of particular predicates such as “=”. For example, equality has to be a reflexive, symmetric and transitive relation. In order to express those properties axiomatically, we need to extend the axiom system of Definition 2.3.3 by two axioms [Dela87, Hami78, Mend64, Schw71]:

(6) For all the terms x , the formula:

$$x = x$$

is an axiom.

(7) If A_1 is the formula derived from the formula A by the substitution of none, some, or all of the occurrences of the term x by the term y , then the formula:

$$(x = y) \rightarrow (A \leftrightarrow A_1)$$

is an axiom.

We have thus axiomatized the reflexivity of equality and a rule of substitution of equal terms. The symmetric and transitive properties of equality are derivable by the axioms (1) to (7) and the rules of generalization and Modus Ponens.

■ 2.3.7

The proof of Theorem 2.3.6 as well as theorems of soundness and completeness for the axiomatic system of Definition 2.3.3, with or without equality, are beyond the context of this book. The reader can find them, for example, in [Klee52, Mend64, RaSi70].

Definition 2.3.8:

- (i) If S is a set of formulae, possibly empty, and if A is a formula of PrL, a **proof of A from S** , denoted by $S \vdash A$, is a finite sequence of formulae B_1, \dots, B_k of PrL, where every B_i , $1 \leq i \leq k$, is either an axiom, or belongs to S , or follows from certain B_j, B_ℓ , $1 \leq j, \ell \leq i$, using the rule of Modus Ponens or the rule of generalization.
- (ii) A is **provable from S** if there exists a proof of A from S .
- (iii) A is **provable** if there exists a proof of A from the empty set. ■ 2.3.8

In PrL, the theorem of deduction is of great interest. The application of the corresponding PL theorem, that is Theorem 1.8.7, to PrL formulae may lead to unexpected results:

Let us assume that we know that there are rich people:

$$\text{rich}(x)$$

There, by the rule of generalization, we have:

$$\text{rich}(x) \vdash (\forall x) \text{rich}(x)$$

And by the “corresponding” PrL deduction theorem we conclude:

$$\text{rich}(x) \rightarrow (\forall x) \text{rich}(x)$$

In other words, the existence of rich people implies that all people are rich which, of course, does not correspond to reality! Therefore, in order to produce correct conclusions, the deduction theorem must include limitations on the use of the rule of generalization.

Theorem 2.3.9: Deduction Theorem:

If S is a set of PrL formulae, A, B PrL formulae such that $S \cup \{A\} \vdash B$, and if the rule of generalization has not been used in the derivation of B from $S \cup \{A\}$ on a free variable occurring in A , then $S \vdash A \rightarrow B$. ■ 2.3.9

The proof of the PrL deduction theorem is similar to the proof of the corresponding PL theorem.

By the rule of Modus Ponens, the converse obviously holds. Then:

$$S \vdash A \rightarrow B \Leftrightarrow S \cup \{A\} \vdash B$$

2.4 Notation in Logic Programming

In section 1.9, we defined several basic concepts of Logic Programming in PL. We will now extend those definitions to PrL [ChLe73, Dela87].

Definition 2.4.1:

- (i) A **literal** is every atom (Definition 2.2.4) or its negation.
- (ii) A sequence of symbols of the form:

$$(\forall x) (\forall x_2) \dots (\forall x_k) (C_1 \vee C_2 \vee \dots \vee C_n)$$

where $C_i, i = 1, \dots, n$ are literals and x_1, \dots, x_k are all the variables occurring in $C_i, 1 \leq i \leq n$, is called a **clause**. If $n = 0$, we have the empty clause, which is denoted by \square . ■ 2.4.1

The parentheses of the quantifiers will be omitted wherever the position of the variables and the quantifiers is explicit.

A clause can equivalently be in one of the following forms.

$$(a) \quad \forall x_1 \dots \forall x_k (A_1 \vee \dots \vee A_m \vee \neg B_1 \vee \dots \vee \neg B_\ell)$$

$$(b) \quad \forall x_1 \dots \forall x_k (A_1 \vee \dots \vee A_m \leftarrow B_1 \wedge \dots \wedge B_\ell)$$

$$(c) \quad \forall x_1 \dots \forall x_k (A_1, \dots, A_m \leftarrow B_1, \dots, B_\ell)$$

- (d) $\{C_1, C_2, \dots, C_n\}$, set-theoretical form, where for all $1 \leq i \leq n$, C_i is A_j , $1 \leq j \leq m$, or C_i is $\neg B_j$, $1 \leq j \leq \ell$
- (e) $A_1, \dots, A_m \leftarrow B_1, \dots, B_\ell$

Each clause is thus a sentence, Definition 2.2.17. The reverse does not hold true, due to the possible existence of the quantifier \exists . However Logic Programming deals with all PrL sentences. We will see how to deal with this problem in section 2.7, when we examine the Skolem Forms.

Example 2.4.2: The following sequences of symbols are clauses:

$$(i) \quad \forall x \forall y \forall z (P(x) \vee \neg Q(x, y) \vee R(x, y, z))$$

$$(ii) \quad \forall x \forall y (\neg P(f(x, y), a) \vee Q(x, y)) \quad \blacksquare \quad 2.4.2$$

Definition 2.4.3: A sentence resulting from the subtraction of the quantifiers from a clause φ and the substitution of all variables by constants, is a **ground instance** of φ . \blacksquare 2.4.3

For example, the sentence

$$P(a) \vee Q(b) \vee \neg R(a, b)$$

is a ground instance of the clause

$$\forall x \forall y (P(x) \vee Q(y) \vee \neg R(x, y))$$

Definition 2.4.4: A **Horn clause** is a clause of the form:

$$\forall x_1 \dots \forall x_k (A \leftarrow B_1, \dots, B_\ell)$$

where A, B_1, \dots, B_ℓ are atoms and $\ell \geq 1$. The atoms B_i , $i = 1, \dots, \ell$, are the assumptions of the Horn clause and A is the conclusion. \blacksquare 2.4.4

Definition 2.4.5: A **goal** is a Horn clause with no conclusion, that is a clause of the form

$$\forall x_1 \dots \forall x_k (\leftarrow B_1, \dots, B_\ell)$$

The atoms B_i are the **subgoals** of the goal. \blacksquare 2.4.5

The intuitive interpretation of a goal becomes clear if we rewrite it in the formal PrL form, and if we use the duality of \forall and \exists as well as De Morgan (see Remark 2.3.4 (3)):

$$\begin{aligned}\forall x_1 \dots \forall x_k (\neg B_1 \vee \dots \vee \neg B_\ell) &\leftrightarrow \forall x_1 \dots \forall x_k \neg (B_1 \wedge \dots \wedge B_\ell) \\ &\leftrightarrow \neg (\exists x_1 \dots \exists x_k) (B_1 \wedge \dots \wedge B_\ell)\end{aligned}$$

In other words, there are no x_1, \dots, x_k such that all the assumptions B_1, \dots, B_ℓ are true.

Definition 2.4.6: A **fact** is a Horn clause with no assumptions, that is a clause of the form:

$$\forall x_1 \dots \forall x_k (A \leftarrow) \quad \blacksquare \quad 2.4.6$$

Remark 2.4.7: In the following sections, we will especially examine sentences of PrL, that is formulae with no free variables, Definitions 2.2.12 and 2.2.16.

There are two reasons which urge us to deal with sentences:

- (1) The clauses, Definitions 2.4.1, 2.4.4, 2.4.5, and 2.4.6, which are used in Logic Programming are sentences of PrL in which all variables are bound by universal quantifiers.
- (2) For every formula φ of PrL which contains exactly the free variables $\{x_1, \dots, x_k\}$, we have:

$$\vdash \varphi \Leftrightarrow \vdash (\forall x_1) \dots (\forall x_k) \varphi$$

(where (\Rightarrow) by the rule of generalization and (\Leftarrow) by axiom (4) and the rule of Modus Ponens).

We thus examine the sentence $(\forall x_1) \dots (\forall x_k) \varphi$ instead of examining the formula φ . \blacksquare 2.4.7

Definition 2.4.8: A **program** is a finite set of Horn clauses. \blacksquare 2.4.8

Let us now illustrate the above concepts with an example.

Example 2.4.9: The following sentences of the English language are given:

- S_1 : Peter is a thief
- S_2 : Mary likes food
- S_3 : Mary likes wine
- S_4 : Peter likes money
- S_5 : Peter likes x if x likes wine
- S_6 : x can steal y if x is a thief and if x likes y

By introducing the constants “Peter”, “Mary”, “wine”, “food”, and “money”, the variables x and y , and the predicates “thief” (arity 1), “likes” (arity 2), and “can_steal” (arity 2), we form the following program:

- C_1 : thief(Peter) \leftarrow
- C_2 : likes(Mary, food) \leftarrow
- C_3 : likes(Mary, wine) \leftarrow
- C_4 : likes(Peter, money) \leftarrow
- C_5 : likes(Peter, x) \leftarrow likes(x , wine)
- C_6 : can_steal(x , y) \leftarrow thief(x), likes(x , y)

Horn clauses C_1 , C_2 , C_3 and C_4 are facts. Horn clauses C_5 and C_6 constitute the procedural part of the program.

Assume we wish to find out what Peter might steal (if there is something which Peter can steal); we have to form the following goal:

$$G : \leftarrow \text{can_steal}(\text{Peter}, y)$$

We will find out the answer in Example 2.10.12.

■ 2.4.9

We will later discuss systematically the ways to derive conclusions from sets of clauses similar to those of the above example.

The theory of interpretations will be developed in the following sections; we will describe methods of assigning truth values to sentences of a PrL language, that is to formulae with no free variables, by means of the interpretations.

2.5 Interpretations of Predicate Logic

Within the context of PL, the finding of the truth value of a compound proposition was based on the concept of **valuation**. By assigning truth values to the atomic formulae of a given proposition A , we determined inductively the truth value of A .

In the following, we will generalize the concept of **interpretation** on which we will ground the development of methods of finding the truth value of a PrL sentence.

Interpretations: an Informal Description

We wish to interpret, that is to assign truth values to PrL sentences. The basic elements of these sentences are terms, namely variables and constants. Therefore we first need to interpret terms by forming a set of objects which constitutes an interpretation of the terms of the sentence. We then can define the truth values of the predicates and the sentences of the language.

Hence, the semantics of PrL are clearly more complex than the semantics of PL. In order to understand fully the concepts which will now be introduced, we will give a few examples.

Example 2.5.1: Assume the language

$$\mathcal{L} = \{Q, \alpha\}$$

where Q is a predicate and α is a constant, and the sentence of this language:

$$S : (\exists x) (\forall y) Q(x, y)$$

Every interpretation of the above language has to determine a set of objects, the elements of which have to correspond to the symbols of \mathcal{L} and assign truth values to the sentences of the languages. We will take \mathbb{N} , the set of natural numbers, as the set of objects. We can now define the interpretation as a structure:

$$\mathcal{A} = (\mathbb{N}, \leq, 1)$$

where:

- \mathbf{N} : the set of natural numbers
- \leq : the relation “less than or equal to” in \mathbf{N}
- 1 : the natural number 1

Based on the interpretation \mathcal{A} , we assign the following correspondences to the symbols of \mathcal{L} :

the symbol Q corresponds to the relation \leq in \mathbf{N} .

the symbol α corresponds to the natural number 1.

The variables of S , x and y , take values from \mathbf{N} . Then the following is the obvious meaning of S in relation to \mathcal{A} :

S : “There exists a natural number x such that,
for every natural number y , $x \leq y$ holds.”

S thus declares that there exists a minimal element in \mathcal{A} and it is obviously true in \mathcal{A} , 1 being the minimal element of \mathcal{A} .

Let us now define a different interpretation for the same language \mathcal{L} .

$$\mathcal{A}' = (\mathbf{N}, >, 1)$$

where Q is the predicate corresponding to the relation “ $>$ ”, the “greater than” relation of the natural numbers, and where α corresponds to 1.

Based on this interpretation, the meaning of S becomes:

S : “There exists a natural number x such that,
for every natural number y , $x > y$ holds.”

S states that there exists a maximal element in \mathcal{A} , and this is obviously not true in \mathcal{A}' , since \mathcal{A}' does not have a maximal element.

In other words, we observe that we can attribute different interpretations for the same language, thus assigning different truth values. ■ 2.5.1

Interpretations and Truth: Formal Description

We will continue with the formal description of the interpretation and the truth of a sentence within the PrL context, [ChLe73, Dela87, Hami78, Meta85, RaSi70].

Definition 2.5.2: Let

$$\mathcal{L} = \{R_0, R_1, \dots, f_0, f_1, \dots, c_0, c_1, \dots\}$$

be a PrL language, where R_i are predicate symbols, f_i are symbols of functions and c_i are constant symbols, $i = 0, 1, \dots$. An **interpretation** of \mathcal{L}

$$\mathcal{A} = (A, \varepsilon(R_0), \varepsilon(R_1), \dots, \varepsilon(f_0), \varepsilon(f_1), \dots, \varepsilon(c_0), \varepsilon(c_1), \dots)$$

consists of:

- (i) a set $A \neq \emptyset$, the universe of the interpretation.
- (ii) an n -ary relation $\varepsilon(R_i) \subseteq A^n$ for every predicate symbol R_i of arity n .
- (iii) a function $\varepsilon(f_i) : A^m \longrightarrow A$ for every function f_i .
- (iv) an element $\varepsilon(c_i) \in A$ for every constant symbol c_i .

$\varepsilon(R_i)$, $\varepsilon(f_i)$ and $\varepsilon(c_i)$ are the **interpretations** of R_i , f_i and c_i respectively in \mathcal{A} .

■ 2.5.2

Example 2.5.3: Let

$$\mathcal{L} = \{=, \leq, +, *, 0, 1\}$$

be the PrL language seen in Example 2.2.2. An interpretation of this language is:

$$\mathcal{A} = (\mathbb{N}, =, \leq, +, *, \dots, 0, 1)$$

where \mathbb{N} is the set of natural numbers, “=” the equality relation in \mathbb{N} , “ \leq ” the relation “less or equal”, and “+”, “*” the addition and the multiplication in \mathbb{N} .

At this point, note that the symbol “+” of \mathcal{L} is just a 3-ary predicate symbol, for example $+(2, 3, 5)$, $2 + 3 = 5$. In the interpretation, this symbol is **interpreted** as $\varepsilon(+)$, the symbol of the natural numbers addition. We usually denote by $\varepsilon(@)$ the object interpreted by the symbol “@” of the language in an interpretation \mathcal{A} .

We then have the following interpretations of symbols of \mathcal{L} :

$$\begin{aligned}
 \varepsilon(=) &\subseteq \mathbb{N} \times \mathbb{N} \\
 \varepsilon(\leq) &\subseteq \mathbb{N} \times \mathbb{N} \\
 \varepsilon(+) &\subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
 \varepsilon(*) &\subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
 \varepsilon(0) &\in \mathbb{N} \\
 \varepsilon(1) &\in \mathbb{N}
 \end{aligned}
 \quad \blacksquare \quad 2.5.3$$

A PrL language and its interpretation thus differ greatly. However, we will often, for the sake of simplicity, treat a language and its interpretation identically. For example, \leq will be used both as a predicate of the language and as an interpretation, instead of $\varepsilon(\leq)$, which is a subset of $\mathbb{N} \times \mathbb{N}$.

As seen in Example 2.5.1, a language can have many interpretations.

Example 2.5.4: A different interpretation of the language \mathcal{L} examined in the previous example is:

$$\mathcal{A}' = (\{2n \mid n \in \mathbb{N}_0\}, =', *', +', 0') \quad (\text{even natural numbers})$$

where

$$\begin{aligned}
 \varepsilon(=) &\text{ is } =', \quad \text{the equality in } \{2n \mid n \in \mathbb{N}_0\} \\
 \varepsilon(*) &\text{ is } *', \quad \text{the multiplication in } \{2n \mid n \in \mathbb{N}_0\} \\
 \varepsilon(+) &\text{ is } +', \quad \text{the addition in } \{2n \mid n \in \mathbb{N}_0\} \\
 \varepsilon(0) &\text{ is } 0', \quad \text{the identity element of } +'
 \end{aligned}$$

As mentioned before, a sentence of a language can be “true” in one of the interpretations of the language and “false” in some other. For example,

$$(\exists y) (\forall x) (x * y = x)$$

denoting the existence of an identity element for $*$, is true in \mathcal{A} and false in \mathcal{A}' .

■ 2.5.4

We will now define inductively the truth of a sentence in an interpretation \mathcal{A} .

Definition 2.5.5: Inductive Definition of the Truth of a Sentence in an Interpretation:

Let

$$\mathcal{L} = \{R_0, R_1, \dots, f_0, f_1, \dots, c_0, c_1, \dots\}$$

be a language and \mathcal{A} one of its interpretations.

- (a) The atomic sentence $R_i(c_{i_1}, c_{i_2}, \dots, c_{i_n})$ is **true** in \mathcal{A} if and only if

$$(\varepsilon(c_{i_1}), \dots, \varepsilon(c_{i_n})) \in \varepsilon(R_i)$$

We then write formally:

$$\mathcal{A} \models R_i(c_{i_1}, \dots, c_{i_n})$$

Let φ , φ_1 , and φ_2 be three sentences of L . Then:

- (b) $\mathcal{A} \models \neg\varphi \Leftrightarrow \mathcal{A} \not\models \varphi$
- (c) $\mathcal{A} \models \varphi_1 \vee \varphi_2 \Leftrightarrow \mathcal{A} \models \varphi_1$ or $\mathcal{A} \models \varphi_2$
- (d) $\mathcal{A} \models \varphi_1 \wedge \varphi_2 \Leftrightarrow \mathcal{A} \models \varphi_1$ and $\mathcal{A} \models \varphi_2$
- (e) $\mathcal{A} \models \varphi_1 \rightarrow \varphi_2 \Leftrightarrow \mathcal{A} \models \varphi_2$ or $\mathcal{A} \not\models \varphi_1$
- (f) $\mathcal{A} \models \varphi_1 \leftrightarrow \varphi_2 \Leftrightarrow (\mathcal{A} \models \varphi_1 \text{ and } \mathcal{A} \models \varphi_2) \text{ or } (\mathcal{A} \not\models \varphi_1 \text{ and } \mathcal{A} \not\models \varphi_2)$
- (g) $\mathcal{A} \models (\exists u)\varphi(u) \Leftrightarrow$ for *some* constant symbol $c \in L$, $\mathcal{A} \models \varphi(c)$
- (h) $\mathcal{A} \models (\forall u)\varphi(u) \Leftrightarrow$ for *all* constant symbols $c \in L$, $\mathcal{A} \models \varphi(c)$

■ 2.5.5

Remark 2.5.6:

- (1) By Definition 2.5.5 and Remark 2.3.7, we have

$$(i) \quad \mathcal{A} \models c_1 = c_2 \Leftrightarrow \varepsilon(c_1) \text{ is identical to } \varepsilon(c_2)$$

- (2) If the atomic sentence $R_i(c_{i_1}, \dots, c_{i_n})$ is true in \mathcal{A} , then R_i takes value t in \mathcal{A} . If it is not true, it takes value f (Exercise 12). ■ 2.5.6

In the above definition, the induction is on the length of the sentences. For example, by (c), every disjunctive sentence $(\varphi_1 \vee \varphi_2)$ is true in \mathcal{A} if and only if at least one of φ_1, φ_2 is true in \mathcal{A} .

All the axioms of Definition 2.3.3 and Remark 2.3.7 are formulae which are true in every interpretation \mathcal{A} . Furthermore, the rules of generalization and Modus Ponens lead from true formulae to true formulae, for all PrL interpretations (soundness of the PrL axiomatic system).

Example 2.5.7: Let $\mathcal{L} = \{Q, f\}$ be a language of arithmetic, and let

$$\mathcal{A} = (\mathbb{Q}_+, =, \varphi) \quad \text{and} \quad \mathcal{A}' = (\mathbb{R}_+, =, g)$$

be two interpretations of \mathcal{L} , where

$$\begin{aligned} \mathbb{Q}_+ &: \text{the set of positive rational numbers} \\ \mathbb{R}_+ &: \text{the set of positive real numbers} \\ g &: g(x) = x^2 \\ \varepsilon'(Q) &: =, \quad \text{the equality in } \mathbb{R}_+ \\ \varepsilon(Q) &: =, \quad \text{the equality in } \mathbb{Q}_+ \\ \varepsilon'(g) &: g \text{ defined in } \mathbb{R}_+ \\ \varepsilon(g) &: g \text{ defined in } \mathbb{Q}_+ \end{aligned}$$

Then the sentence $S : (\forall x)(\exists y) Q(x, g(y))$, namely $(\forall x)(\exists y)(x = y^2)$, is true in \mathcal{A}' but not in \mathcal{A} since the equation $x = y^2$, where x is given and positive, always has a solution in \mathbb{R}_+ , but not always in \mathbb{Q}_+ . ■ 2.5.7

Example 2.5.8: For the language $\mathcal{L} = \{Q, f, a, b\}$, we can define the following interpretation:

$$\mathcal{A} = (A, \text{child}, \text{mother}, \text{John}, \text{Mary}, \text{Napoleon})$$

where:

$$\begin{aligned} A &: \text{the set of all human beings} \\ \varepsilon(Q) &: \text{the relation "child",} \\ &\quad \text{i.e., } \varepsilon(Q)(x_1, x_2) = \text{child}(x_1, x_2) = \text{"}x_1 \text{ is the child of } x_2\text{"} \\ \varepsilon(f) &: \text{the relation "mother",} \\ &\quad \text{i.e., } \varepsilon(f)(x) = \text{mother}(x) = \text{"the mother of } x\text{"} \\ \varepsilon(a) &: \text{the person Mary} \\ \varepsilon(b) &: \text{the person John} \\ \varepsilon(c) &: \text{the person Napoleon} \end{aligned}$$

We observe that, with \mathcal{A} , one symbol which does not occur as an element of the language, namely c , is never the less interpreted. We will see in the following Theorem 2.5.14, that such an interpretation of symbols which do not belong to the language is not problematic.

Let $S : Q(b, f(b)) \vee (\exists x) (Q(a, x))$. According to \mathcal{A} , the interpretation of S is:

$S :$ “John is the child of the mother of John

or

there exists a person x such that Mary is a child of x ”.

John is obviously the child of the mother of John. S is therefore true in the interpretation \mathcal{A} (case (e) of Definition 2.5.5). ■ 2.5.8

Definition 2.5.9: Let \mathcal{L} be a language and σ a sentence. The interpretation \mathcal{A} of \mathcal{L} is a **model** of σ if and only if $\mathcal{A} \models \sigma$. ■ 2.5.9

Definition 2.5.10: If \mathcal{L} is a language and \mathcal{A} an interpretation of \mathcal{L} , then the set

$$\theta(\mathcal{A}) = \{\sigma \mid \mathcal{A} \models \sigma\}$$

is the **theory of the interpretation**. ■ 2.5.10

In other words, the theory of an interpretation is the set of all the sentences which are true in this interpretation.

In the previous example we saw that it is possible to interpret symbols which do not occur as elements of the language. We will give a definition which is useful when dealing with such situations.

Definition 2.5.11: Let us assume we have a language \mathcal{L} and an interpretation \mathcal{A} of \mathcal{L} , such that the elements a_1, a_2, \dots of the universe of \mathcal{A} are not interpretations of constant symbols of \mathcal{L} .

(1) We form a *new language*:

$$\mathcal{L}^* = \mathcal{L} \cup \{c_1, c_2, \dots\}$$

where symbols c_1, c_2, \dots are *new* constant symbols which do not belong to \mathcal{L} . \mathcal{L}^* is said to be an **elementary extension** of \mathcal{L} .

(2) If we have an interpretation

$$\mathcal{A} = (A, R_1, R_2, \dots, P_1, P_2, \dots, d_1, d_2, \dots)$$

then we can form a *new interpretation*:

$$\mathcal{A}^* = (A, R_1, R_2, \dots, P_1, P_2, \dots, d_1, d_2, \dots, a_1, a_2, \dots)$$

of the language $\mathcal{L}^* = \mathcal{L} \cup \{c_1, c_2, \dots\}$, where $c_1, c_2, \dots \notin \mathcal{L}$, thus assigning a suitable interpretation to c_1, c_2, \dots , and actually imposing $\varepsilon(c_1) = a_1, \varepsilon(c_2) = a_2, \dots$. \mathcal{A}^* is then called an **elementary extension** of \mathcal{A} . ■ 2.5.11

Example 2.5.12: Assume

$$\mathcal{L} = \{=, \leq, +, *, 1, 0\} \quad \text{and} \quad \mathcal{A} = (\mathbb{N}, =, \leq, +, *, 0, 1)$$

We can then form the language:

$$\mathcal{L}^* = \{=, \leq, +, *, 0, 1, c_1, c_2, \dots\}$$

and its interpretation

$$\mathcal{A}^* = (\mathbb{N}, =, \leq, +, *, 0, 1, 2, 3, \dots) \quad \text{■ 2.5.12}$$

Remark 2.5.13: We have already enriched the language \mathcal{L} with constants; \mathcal{L} can however also be enriched with new symbols of functions and predicates. \mathcal{L}^* , the extension of \mathcal{L} with new symbols of functions and predicates, is a non-elementary extension of \mathcal{L} . The corresponding extension of \mathcal{A} , namely \mathcal{A}^* , is a non-elementary extension of \mathcal{A} . ■ 2.5.13

Here is a theorem about the verifiability of a sentence within the PrL context.

Theorem 2.5.14: Let \mathcal{L} be a language, \mathcal{A} an interpretation of \mathcal{L} and \mathcal{L}^* and \mathcal{A}^* their respective elementary extensions such that all the elements of the universe of \mathcal{A}^* are interpretations of symbols of \mathcal{L}^* . Let σ be a sentence of \mathcal{L} . Then σ is true in \mathcal{A} if and only if it is true in \mathcal{A}^* . Formally

$$\mathcal{A} \models \sigma \Leftrightarrow \mathcal{A}^* \models \sigma$$

Proof: By induction on the length of σ .

If σ is a sentence $P(c_1, \dots, c_k)$, then

$$\begin{aligned}
 \mathcal{A}^* \models P(c_1, \dots, c_k) &\Leftrightarrow (\varepsilon(c_1), \dots, \varepsilon(c_k)) \in \varepsilon(P) \\
 &\quad \text{and } \varepsilon(c_1), \dots, \varepsilon(c_k) \in \mathcal{A}^*, \quad \varepsilon(P) \subseteq (\mathcal{A}^*)^k \\
 &\Leftrightarrow (\varepsilon(c_1), \dots, \varepsilon(c_k)) \in \varepsilon(P) \\
 &\quad \text{and } \varepsilon(c_1), \dots, \varepsilon(c_k) \in \mathcal{A}, \quad \varepsilon(P) \subseteq \mathcal{A}^*, \\
 &\quad \text{since } \sigma \text{ is a sentence of } \mathcal{L} \\
 &\Leftrightarrow \mathcal{A} \models P(c_1, \dots, c_k)
 \end{aligned}$$

The cases $\sigma : \neg\varphi$, $\sigma : \varphi \vee \varphi'$, $\sigma : \varphi \wedge \varphi'$, and $\sigma : \varphi \rightarrow \varphi'$, are treated similarly.

Let us assume that σ is the sentence $(\exists x)\varphi(x)$, where x is the only free variable of σ . Then:

$$\begin{aligned}
 \mathcal{A}^* \models (\exists x)\varphi(x) &\Leftrightarrow \text{for a constant symbol } c \text{ of } \mathcal{L}^*, \mathcal{A}^* \models \varphi(c) \\
 &\Leftrightarrow \mathcal{A} \models \varphi(c) \text{ due to the induction assumption} \\
 &\quad \text{and to the fact that } \varphi \text{ is a sentence of } \mathcal{L}.
 \end{aligned}$$

The case $\sigma : (\forall x)\varphi(x)$ is examined similarly. ■ 2.5.14

According to Theorem 2.5.14, the truth of a sentence in an interpretation \mathcal{A}^* does not depend on the selection of the new constant symbols and their interpretations.

Definition 2.5.15: A sentence σ of a language \mathcal{L} is **verifiable** or **satisfiable** or **consistent** if and only if there is an interpretation \mathcal{A} of \mathcal{L} in which it is true, i.e., $\mathcal{A} \models \sigma$. ■ 2.5.15

Definition 2.5.16: A set of sentences S is **verifiable** or **satisfiable** or **consistent** if there exists an interpretation in which all the sentences of S are true. In the opposite case, S is a **non-verifiable** or **non-satisfiable**, or **inconsistent** set. ■ 2.5.16

Example 2.5.17:

Assume

$$\mathcal{A} = (\mathbb{N}, =, \leq, +, *, 0, 1)$$

Then $\mathcal{A} \models \sigma$, where σ is any one of the sentences (1) - (4) of Example 2.2.8.

We will prove that $\mathcal{A} \models (\forall x)(\forall y)(x = y \rightarrow y = x)$.

Assume $\mathcal{A} \models (\forall x)(\forall y)(x = y \rightarrow y = x)$ does not hold true. Then by Definition 2.5.5 (h) and (b) there exists c_1, c_2 such that

$$\mathcal{A} \models \neg[(c_1 = c_2) \rightarrow (c_2 = c_1)]$$

Then

$$\mathcal{A} \models \neg[\neg(c_1 = c_2) \vee (c_2 = c_1)]$$

or

$$\mathcal{A} \models [(c_1 = c_2) \wedge \neg(c_2 = c_1)]$$

and by Definition 2.5.5 (d)

$$\mathcal{A} \models (c_1 = c_2) \tag{1}$$

and

$$\mathcal{A} \models \neg(c_2 = c_1) \tag{2}$$

By the seventh axiom of Remark 2.3.7, i.e., the axiom of substitution of equal terms, we use (1) to substitute c_2 for the value of c_1 in (2). Then

$$\mathcal{A} \models \neg(c_2 = c_2)$$

or by Definition 2.5.5 (g)

$$\mathcal{A} \models (\exists x) \neg(x = x)$$

and by the duality of \forall and \exists we have

$$\mathcal{A} \models \neg(\forall x)(x = x)$$

which is a contradiction, since it goes against reflexivity of equality. ■ 2.5.17

Example 2.5.18: The standard interpretation of $\mathcal{L} = \{=, \leq, +, *, 0, 1\}$ is:

$$\mathcal{A} = (\mathbb{N}, =, \leq, +, *, 0, 1)$$

Another interpretation of \mathcal{L} , where \mathbb{Q} is the set of rational numbers, is:

$$\mathcal{B} = (\mathbb{Q}, =, \leq, +, *, 0, 1)$$

Let σ be the sentence (actually the formal definition of a dense order):

$$(\forall x)(\forall y) [\neg(x = y) \rightarrow (\exists z) [\neg(z = x) \wedge \neg(z = y) \wedge (x \leq z) \wedge (z \leq y)]]$$

Then, $\mathcal{A} \not\models \sigma$ but $\mathcal{B} \models \sigma$.

■ 2.5.18

Just as in PL, there are sentences in every language of PrL which are true in all interpretations of the language.

Definition 2.5.19: Logically True Formula:

If the formula σ of the language \mathcal{L} is true in every interpretation of \mathcal{L} , then σ is **logically true**.

■ 2.5.19

Example 2.5.20: Let φ be a sentence of \mathcal{L} and ψ a formula of \mathcal{L} .

Then the sentence

$$S: ((\forall v)(\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow (\forall v)\psi))$$

is true in every interpretation of \mathcal{L} .

Proof: Assume that S is not true in all interpretations of \mathcal{L} . There then exists an interpretation \mathcal{A} of \mathcal{L} such that:

$$\mathcal{A} \not\models (\forall v)(\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow (\forall v)\psi)$$

Then by Definition 2.5.5, we have

$$\text{not } [\mathcal{A} \not\models (\forall v)(\varphi \rightarrow \psi)] \quad \text{or} \quad \mathcal{A} \models \varphi \rightarrow (\forall v)\psi$$

and by De Morgan:

$$\text{not } [\mathcal{A} \not\models (\forall v)(\varphi \rightarrow \psi)] \quad \text{and} \quad \text{not } [\mathcal{A} \models \varphi \rightarrow (\forall v)\psi]$$

From the Law of Double Negation we take:

$$\mathcal{A} \models (\forall v)(\varphi \rightarrow \psi) \quad (1) \quad \text{and} \quad \mathcal{A} \not\models \varphi \rightarrow (\forall v)\psi \quad (2)$$

Then $\mathcal{A} \not\models \neg\varphi \vee (\forall v)\psi$ follows from (2), and by Definition 2.5.5 (b), we have:

$$\mathcal{A} \models \varphi \quad \text{and} \quad \mathcal{A} \not\models (\forall v)\psi \quad (3)$$

By (3) and Definition 2.5.5 (h) we know that there exists a $c \in \mathcal{L}$, such that:

$$\mathcal{A} \models \neg\psi(v/c)$$

and from $\mathcal{A} \models \varphi$ and Definition 2.5.5 (d) we take

$$\mathcal{A} \models \varphi \wedge \neg\psi(v/c)$$

or equivalently

$$\mathcal{A} \models \neg(\neg\varphi \vee \psi(v/c))$$

or equivalently

$$\mathcal{A} \models \neg((\varphi \rightarrow \psi)(v/c)) \quad (4)$$

By (1) and Definition 2.5.5 (h) for every $c \in \mathcal{L}$

$$\mathcal{A} \models (\varphi \rightarrow \psi)(v/c) \quad (5)$$

holds. However φ is a sentence, and it has no free variables. Hence there is no possible substitution in φ . Then (5) takes the form $\mathcal{A} \models \varphi \rightarrow \psi(v/c)$ for all $c \in \mathcal{L}$, and that is a contradiction by (4). S is hence true in all the interpretations of \mathcal{L} .

■ 2.5.20

Definition 2.5.21: Let \mathcal{L} be a PrL language and S a set of PrL sentences. The sentence σ is a **consequence** of S , formally $S \models \sigma$, if and only if every interpretation \mathcal{A} of \mathcal{L} verifying all the propositions of S , verifies σ . This is denoted by:

$$\sigma \in \text{Con}(S) \Leftrightarrow (\forall \mathcal{A}) [\mathcal{A} \models S \Rightarrow \mathcal{A} \models \sigma] \quad \blacksquare \text{ 2.5.21}$$

2.6 Normal Forms in Predicate Logic

In PL, we examined two equivalent forms of a proposition; the CNF (Conjunctive Normal Form) and the DNF (Disjunctive Normal Form). Sentences of analogous forms are also found in PrL. Within the context of PrL, however, there are two additional forms; the Prenex Normal Form (PNF) and the Skolem Normal Form (SNF), depending on the quantifiers occurring in the sentence [Chur56, ChLe73, Hami78, Klee52, Thay88]. By reducing two propositions into one of the above

forms, we can easily compare them and determine whether they are equivalent, whether one of them is the negation of the other, or, simply, whether there is anything significant about them. The Skolem Normal Forms have an important role in Logic Programming.

We will examine analytically these forms in the following subsections.

Definition 2.6.1: The formula φ is a **prenex formula**, PNF for short, if φ is of the form:

$$\varphi : (Q_1x_1)(Q_2x_2)\dots(Q_nx_n)\sigma$$

where every Q_i , $i = 1, \dots, n$ is one of the quantifiers \forall , \exists , and σ is a formula without quantifiers. $(Q_1x_1)\dots(Q_nx_n)$ is called the **prefix** of φ , and σ is called the **matrix** of φ . ■ 2.6.1

Example 2.6.2: The sentences below are in a PNF:

$$(i) \quad (\forall x)(\forall y)[P(x, y) \rightarrow Q(x)]$$

$$(ii) \quad (\forall x)(\exists y)[Q(x, y) \vee P(x, y)] \quad \blacksquare \quad 2.6.2$$

To reduce a formula or a sentence to a PNF, besides the formulae used in PL, we also use in PrL the following formulae:

$$(1) \quad (Qx)P(x) \vee G \leftrightarrow (Qx)[P(x) \vee G] \quad \text{where } x \text{ does not occur free in } G$$

$$(2) \quad (Qx)P(x) \wedge G \leftrightarrow (Qx)[P(x) \wedge G] \quad \text{where } x \text{ does not occur free in } G$$

$$(3) \quad \neg(\forall x)P(x) \leftrightarrow (\exists x)(\neg P(x))$$

$$(4) \quad \neg(\exists x)P(x) \leftrightarrow (\forall x)(\neg P(x))$$

$$(5) \quad (\forall x)P(x) \wedge (\forall x)G(x) \leftrightarrow (\forall x)(P(x) \wedge G(x))$$

$$(6) \quad (\exists x)P(x) \vee (\exists x)G(x) \leftrightarrow (\exists x)(P(x) \vee G(x))$$

where (Qx) is $(\forall x)$ or $(\exists x)$.

The above formulae are derivable in the PrL axiomatic system (Theorem 2.3.6). Formulae (1) and (2) declare that the scope of action of the quantifiers includes conjunctions and disjunctions, provided that the variables of the quantifiers introduced do not occur as free variables. Formulae (3) and (4) are obvious cases,

considering the duality of \forall and \exists . Cases (5) and (6) declare that the universal quantifier is distributive on conjunctions whereas the existential quantifier is distributive on disjunctions. The reverse however does not hold true:

$$(7) \quad (\exists x) P(x) \wedge (\exists x) G(x) \not\leftrightarrow (\exists x) (P(x) \wedge G(x))$$

$$(8) \quad (\forall x) P(x) \vee (\forall x) G(x) \not\leftrightarrow (\forall x) (P(x) \vee G(x))$$

In order to obtain equivalences in cases similar to (7) and (8), we first rename all the occurrences of x in the formulae $(\forall x) G(x)$ and $(\exists x) G(x)$. And we are allowed to do just so since x is bound. Then, by Theorem 2.3.6, and (5) and (6), we obtain (how?) the following equivalences:

$$(9) \quad (Q_1 x) P(x) \wedge (Q_2 x) G(x) \leftrightarrow (Q_1 x) (Q_2 z) [P(x) \wedge G(z)]$$

$$(10) \quad (Q_1 x) P(x) \vee (Q_2 x) G(x) \leftrightarrow (Q_1 x) (Q_2 z) [P(x) \vee G(z)]$$

where $Q_1, Q_2 \in \{\forall, \exists\}$, and where the variable z does not occur in $G(x)$ at all, and it does not occur as a free variable in $P(x)$ in (9) and (10).

We will now describe the formal procedure for the reduction of a sentence to a PNF:

Construction 2.6.3:

Step 1:

Elimination of the symbols \leftrightarrow and \rightarrow by means of the formulae:

$$(1a) \quad (A \leftrightarrow B) \leftrightarrow ((A \rightarrow B) \wedge (B \rightarrow A))$$

$$(1b) \quad (A \rightarrow B) \leftrightarrow (\neg A \vee B)$$

Step 2:

Transfer of the negations in front of the atoms by means of the formulae:

$$(2a) \quad \neg(A \vee B) \leftrightarrow \neg A \wedge \neg B$$

$$(2b) \quad \neg(A \wedge B) \leftrightarrow \neg A \vee \neg B$$

$$(2c) \quad \neg(\neg A) \leftrightarrow A$$

$$(2d) \quad \neg(\forall x) A \leftrightarrow (\exists x) \neg A$$

$$(2e) \quad \neg(\exists x) A \leftrightarrow (\forall x) \neg A$$

Step 3:

Transfer of the quantifiers to the left by means of the formulae:

$$(3a) \quad (\forall x) A(x) \wedge (\forall x) B(x) \leftrightarrow (\forall x) [A(x) \wedge B(x)]$$

$$(\exists x) A(x) \vee (\exists x) B(x) \leftrightarrow (\exists x) [A(x) \vee B(x)]$$

$$(3b) \quad (Qx) A(x) \wedge B \leftrightarrow Q(x) [A(x) \wedge B]$$

$$(Qx) A(x) \vee B \leftrightarrow Q(x) [A(x) \vee B]$$

where x does not occur as a free variable in B in the formulae (3b).

$$(3c) \quad (Q_1x) A(x) \vee (Q_2x) B(x) \leftrightarrow (Q_1x) (Q_2z) [A(x) \vee B(z)]$$

$$(Q_1x) A(x) \wedge (Q_2x) B(x) \leftrightarrow (Q_1x) (Q_2z) [A(x) \wedge B(z)]$$

where x does not occur as a free variable in B in the formulae (3c), z does not occur in B and not as a free variable in A , and $B(z)$ is the result of the replacement of every free occurrence of x by z . ■ 2.6.3

Example 2.6.4:

$$\varphi : (\forall x) P(x) \rightarrow (\exists x) R(x) \leftrightarrow \neg(\forall x) P(x) \vee (\exists x) R(x) \quad (1b)$$

$$\leftrightarrow (\exists x) \neg P(x) \vee (\exists x) R(x) \quad (2d)$$

$$\leftrightarrow (\exists x) [\neg P(x) \vee R(x)] \quad (3a)$$

■ 2.6.4

Example 2.6.5:

$$\varphi : (\forall x) (\forall y) [(\exists z) (P(x, z) \wedge P(y, z)) \rightarrow (\exists u) R(x, y, u)]$$

$$\leftrightarrow (\forall x) (\forall y) [\neg(\exists z) (P(x, z) \wedge P(y, z)) \vee (\exists u) (R(x, y, u))] \quad (1b)$$

$$\leftrightarrow (\forall x) (\forall y) [(\forall z) (\neg P(x, z) \vee \neg P(y, z)) \vee (\exists u) R(x, y, u)] \quad (2b)$$

$$\leftrightarrow (\forall x) (\forall y) (\forall z) [\neg P(x, z) \vee \neg P(y, z) \vee (\exists u) R(x, y, u)] \quad (3b)$$

$$\leftrightarrow (\forall x) (\forall y) (\forall z) (\exists u) [\neg P(x, z) \vee \neg P(y, z) \vee R(x, y, u)] \quad (3b)$$

■ 2.6.5

We can now move to the conversion of a sentence φ to a sentence φ^* which is **universal**, that is, containing only universal quantifiers, and in a Prenex Form, such that φ^* is verifiable if and only if φ is verifiable.

The Skolem Normal Form

Theorem 2.6.6: Loewenheim, Skolem:

For every sentence φ of PrL, we can form a universal sentence φ^ such that:*

$$\varphi \text{ verifiable} \Leftrightarrow \varphi^* \text{ verifiable} \quad \blacksquare \text{ 2.6.6}$$

We will now describe how to form φ^* , φ being an arbitrary PrL sentence:

Definition 2.6.7: Let φ be a sentence of a PrL language.

Step 1 :

We determine the PNF of φ .

Step 2 :

We gradually cross out every existential quantifier $(\exists y)$, replacing all the occurrences of y by a new, so far unused, function symbol f of all the variables bound by universal quantifiers which precede $(\exists y)$. f is called a **Skolem function**. The sentence φ^* obtained after the application of steps 1 and 2 is a **Skolem Normal Form**, of φ , SNF for short. \blacksquare 2.6.7

Example 2.6.8: Assume the sentence

$$\varphi : (\forall x) (\exists y) (\forall z) (\exists v) P(x, y, v)$$

which is in a PNF.

- (1) We cross out $(\exists y)$ and replace y with the Skolem function $f(x)$. We thus obtain the sentence:

$$\varphi_1 : (\forall x) (\forall y) (\exists v) P(x, f(x), z, v)$$

- (2) We cross out $(\exists v)$ in φ_1 and replace v with the Skolem function $g(x, z)$, since $(\forall x)$, $(\forall z)$ precede $(\exists v)$. We thus obtain

$$\varphi^* : (\forall x) (\forall z) P(x, f(x), z, g(x, z)) \quad \blacksquare \text{ 2.6.8}$$

Intuitively, in the above example, $f(x)$ indicates the existence of a variable y which depends on x . (By Definition 2.5.5 (h) and (g), there is a constant for each x which is likely to replace y). So y is an existential variable, in other words it is bound by an existential quantifier. Thus φ_1 and φ^* imply φ by means of $f(x)$ and $g(x, z)$.

Example 2.6.9: The SNF of

$$\varphi : (\exists y) (\forall x) (\forall z) \psi(x, y, z)$$

is

$$\varphi^* : (\forall x) (\forall z) \psi(x, c, z)$$

where c is a constant since the corresponding Skolem function f has no variables ($(\exists y)$ is the first quantifier of φ , hence y is not determined by any variable and f is the constant function c). ■ 2.6.9

The reduction of a sentence to an SNF is of great importance to Logic Programming, this will become even more evident when we present the Resolution Proof Method of PrL.

In PL, Definition 1.9.4, we saw that a sentence of the form:

$$(A_{1_1} \vee \dots \vee A_{1_n}) \wedge \dots \wedge (A_{k_1} \vee \dots \vee A_{k_n})$$

namely a conjunction of disjunctions, can be set-theoretically represented as a set of the form

$$\{\{A_{1_1}, \dots, A_{1_n}\}, \dots, \{A_{k_1}, \dots, A_{k_n}\}\}$$

Keeping in mind Definition 2.4.1 as well as the quantifiers, we now present this definition within the context of PrL.

Definition 2.6.10: Let φ be a PrL sentence in an SNF:

$$\varphi : (\forall x_1) \dots (\forall x_\ell) A(x_1, \dots, x_\ell) \quad (*)$$

where $A(x_1, \dots, x_\ell)$ is the conjunction:

$$C_1(x_1, \dots, x_\ell) \wedge \dots \wedge C_k(x_1, \dots, x_\ell)$$

and every C_i , $1 \leq i \leq k$ is a disjunction of the atoms or the negations of the atoms P_{i_1}, \dots, P_{i_p} of PrL. Then φ is **set-theoretically** represented as a set

$$S = \{\{P_{i_1}, \dots, P_{i_n}\}, \dots, \{P_{k_1}, \dots, P_{k_n}\}\} \quad \text{or} \quad S = \{C_1, \dots, C_k\}$$

Every C_i is a **clause**, S being the set of clauses. ■ 2.6.10

Example 2.6.11: The sentence

$$(\forall x)(\forall z)[P_1(x, z) \wedge (P_2(x) \vee P_3(z)) \wedge P_4(z, x)]$$

has, as its set-theoretical form, the set:

$$S = \{\{(P_1(x, z)), \{P_2(x), P_3(z)\}, \{P_4(z, x)\}\}\} \quad \text{■ 2.6.11}$$

In the following chapters, we will present **Herbrand interpretations** as well as proof methods of the verifiability of sentences or sets of clauses of PrL.

2.7 Herbrand Interpretations

In this section, we will describe a special kind of interpretation, the Herbrand interpretations [ChLe73, Dela87, Klee52, Thay88] which have a catalytic role in the theoretical foundation of Logic Programming.

Herbrand interpretations were the subject of J. Herbrand's thesis in 1930. There is probably no exaggeration in saying that without Herbrand's contribution, Logic Programming might still be a far-off dream. The basic problem in Automatic Theorem Proving is the determination of a general procedure by means of which we can prove whether a PrL sentence is true or not [ChLe73]. In 1936, Turing and Church, both working on their own, proved that there is no such general procedure. Herbrand had however already solved the problem indirectly, by giving an algorithm for the construction of an interpretation refuting a given formula φ . If φ is true, there is no interpretation refuting it and the algorithm stops after a finite number of steps.

The first attempts to use Herbrand's ideas in Logic Programming go back to 1960 and must be credited to Gilmore as well Davis and Putnam; these attempts were however not truly successful. Success came in 1965 with Robinson's application of Herbrand's method, due to the introduction and use of resolution.

Description of the Herbrand Universe

Given a PrL sentence φ , we want to be able to determine whether φ is verifiable or not. We thus decide about a possible verifiability of the sentence by examining properly the corresponding set of clauses S of φ .

However the proof of the verifiability or non-verifiability of all the basic terms occurring in a clause is almost impossible. We therefore create a set, a universe, in which the terms of φ take values. This set is called the Herbrand Universe. The construction of the **Herbrand Universe** is carried out inductively as follows:

Construction 2.7.1: Construction of the Herbrand Universe:

Let S be the set of clauses corresponding to the sentence φ .

Step 1 :

$$H_0 = \begin{cases} \{c \mid c \text{ constant occurring in } S\} \\ \{c_0\} \text{ if } S \text{ does not contain any constant.} \end{cases}$$

c_0 is a new constant (new; meaning that it does not occur in S), which we introduce arbitrarily.

Step $i + 1$:

$$H_{i+1} = H_i \cup \left\{ f(a_1, \dots, a_n) \mid a_j, 1 \leq j \leq n, \text{ are terms of } H_i, \text{ and } \right. \\ \left. f \text{ a function or a constant occurring in } S \right\}$$

Finally, we impose:

$$H = \cup_{i \in \mathbb{N}} H_i$$

Then H , the set of all terms formed with the constants of H_0 and the functions occurring in S , is called the **Herbrand Universe** for S . The H_i , $i = 0, 1, 2, \dots$, are called the **Herbrand sets** of S . ■ 2.7.1

Example 2.7.2: Let a be a constant and S the set of clauses:

$$S = \{\{P(a)\}, \{\neg P(a), P(f(x))\}\}$$

Then:

$$\begin{aligned} H_0 &= \{a\} \\ H_1 &= \{a, f(a)\} \\ H_2 &= \{a, f(a), f(f(a))\} \\ &\dots \end{aligned}$$

and finally

$$H = \{a, f(a), f(f(a)), f(f(f(a))), \dots\} \quad \blacksquare \quad 2.7.2$$

Example 2.7.3: Assume the program:

$$\begin{aligned} P(x) &\leftarrow Q(f(x), g(x)) \\ R(x) &\leftarrow \end{aligned}$$

We introduce the constant a and have:

$$H = \{a, f(a), g(a), f(f(a)), f(g(a)), g(f(a)), g(g(a)), f(f(f(a))), \dots\} \quad \blacksquare \quad 2.7.3$$

Example 2.7.4:

$$S = \{\{P(x), Q(x)\}, \{T(x), \neg R(x)\}\}$$

We introduce the new constant c_0 . Then $H_0 = \{c_0\}$. Since there are no function symbols occurring in S , we have $H = H_0 = \{c_0\}$. $\blacksquare \quad 2.7.4$

In theory, in order to determine whether a sentence is verifiable, we must find out if there is any interpretation (in an eventual infinite set of interpretations) verifying it. Things become much easier if we limit ourselves to universal sentences, i.e., sentences using only universal quantifiers; in that case, we only need to work on a small group of interpretations called **Herbrand interpretations**.

We will now give the formal description of a Herbrand interpretation.

Definition 2.7.5: Herbrand Interpretation:

Let S be a set of clauses and H the Herbrand universe corresponding to S . An **Herbrand interpretation** \mathcal{A}_H for S is defined as follows:

- (i) H is the universe of the interpretation.
- (ii) The interpretation of every constant symbol is the constant itself.
- (iii) The interpretation of every term $f(t_1, \dots, t_n)$, where f is a function or a constant, is $f(t'_1, \dots, t'_n)$, where t'_1, \dots, t'_n are the interpretations of t_1, \dots, t_n respectively.
- (iv) The interpretation of every n -ary predicate symbol $P(t_1, \dots, t_n)$ is an n -ary relation $P(t'_1, \dots, t'_n)$ in H , where t'_1, \dots, t'_n are the interpretations of t_1, \dots, t_n respectively. ■ 2.7.5

Example 2.7.6: Let $\mathcal{L} = \{\leq, +, *, s, 0\}$ be a language of Arithmetic, where s is the successor function. We have:

$$\begin{aligned} H_0 &= \{0\} \\ H_1 &= \{0, s(0)\} \\ H_2 &= \{0, s(0), s(s(0))\} \\ &\dots \end{aligned}$$

and finally

$$H = \{0, s(0), s(s(0)), s(s(s(0))), \dots\}$$

The interpretations of the elements of \mathcal{L} different from 0 are the corresponding relations in H . For example, for s and \leq , we have according to Definition 2.5.2.

$$\varepsilon(s) : H \mapsto H = a \mapsto s(a) \in H$$

$$\varepsilon(\leq) \subseteq H^2 : \varepsilon(\leq) = \{(0, s(0)), (s(0), s^2(0)), \dots, (s^n(0), s^{n+1}(0)), \dots\}$$

■ 2.7.6

We next give a basic theorem of the Herbrand interpretations.

Theorem 2.7.7: *Let φ be a universal sentence, S being its set of clauses. Then φ is verifiable (in some interpretations) if and only if it is verifiable in a Herbrand interpretation.*

Proof:

(\Leftarrow) : A Herbrand interpretation is an interpretation (trivial case).

(\Rightarrow) : We wish to prove here that if φ is verifiable in an interpretation \mathcal{A} , then we can define relations in the Herbrand universe which satisfy the clauses of S . Let us therefore assume that φ is verified in an interpretation \mathcal{A} with A as its universe. In order to denote the interpretations of the various symbols of \mathcal{L} , i.e., of the language to which φ belongs, we use:

$$\varepsilon(c) = \hat{c} \in A, \text{ for every constant symbol } c$$

$$\varepsilon(f) = \hat{f} : A^n \longrightarrow A, \text{ for every function of } n \text{ variables}$$

$$\varepsilon(t) = \varepsilon(f(t_1, \dots, t_n)) = \hat{f}(\hat{t}_1, \dots, \hat{t}_n), \text{ for every term } t = f(t_1, \dots, t_n)$$

$$\varepsilon(R) = \hat{R} \subseteq A^n, \text{ for every } n\text{-ary predicate } R.$$

For every n -ary predicate symbol R we define a relation R_H in H (Definition 2.7.5, Definition 2.5.2 (ii), Definition 2.5.5) as follows:

$$R_H(t_1, \dots, t_n) \Leftrightarrow (\hat{t}_1, \dots, \hat{t}_n) \in \hat{R}$$

We thus have a Herbrand interpretation \mathcal{A}_H .

Let us impose $A' = \{t \mid t \in H\}$. The structure \mathcal{A}' , with A' as its universe and with the restrictions of the relations of \mathcal{A} in A' is obviously an interpretation of S . Furthermore, $\mathcal{A} \models \varphi$, and whatever occurs in φ , occurs by definition in A . Hence $\mathcal{A}' \models \varphi$. Then, by the definition of \mathcal{A}_H , $\mathcal{A}_H \models \varphi$.

Hence φ is indeed verifiable in a Herbrand interpretation. ■ 2.7.7

By Theorem 2.7.7, if φ is not verifiable in a Herbrand interpretation, then φ is not verifiable; there is no interpretation satisfying φ .

In other words:

By means of the Herbrand interpretations, we reduce the non-verifiability of a set of clauses to the non-verifiability of a set of ground instances of those clauses in a Herbrand Universe. Since there are no variables occurring in any ground instance of the clauses, verifiability can be proved by means of PL methods just like semantic tableau and resolution.

Beth-proofs, as well as resolution, are algorithmic proofs (as opposed to the usual method for verifiability in PrL, Example 2.5.17). This result is known as the Herbrand theorem, no matter what form it may take in the classical or modern bibliography. It will be analysed in a following section dealing with proofs by means of semantic trees.

2.8 Proofs with Systematic Tableaux

In PrL, just as in PL, we can determine whether or not a sentence or a set of sentences are satisfiable. The methods which we have already examined can be used within the PrL context.

Let us start with the semantic tableaux [Fitt69, Fitt90, Meta85, Smul68]. These tableaux are used in the finding of the truth value of a compound sentence of a PrL language \mathcal{L} .

Definition 2.8.1: Proofs with Complete Systematic Tableaux:

Assume we have a language \mathcal{L} and that c_0, c_1, \dots are its constant symbols constituting a list of constants. (The meaning of this list will appear clearly in Construction 2.8.5).

Let $\sigma, \sigma_1, \sigma_2$ be sentences of \mathcal{L} . The semantic tableaux are given in the table on the facing page. ■ 2.8.1

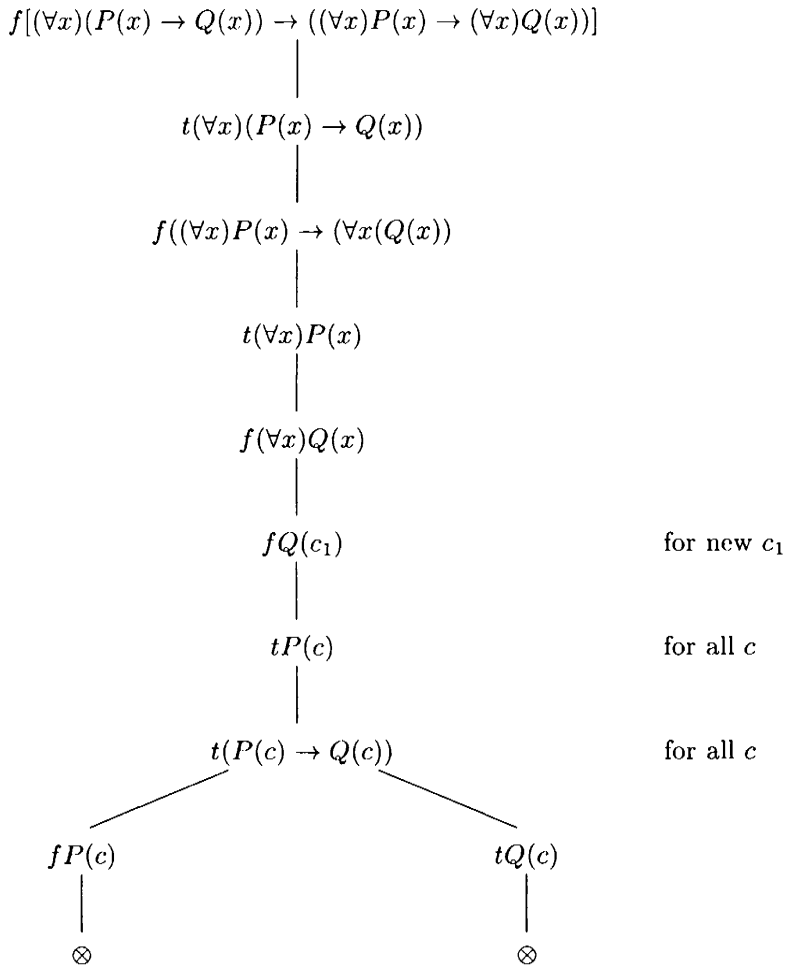
The PrL semantic tableaux are extensions of the corresponding PL tableaux with additional cases for the quantifiers.

With the semantic tableau

$$\begin{array}{c} t(\forall x)\varphi(x) \\ | \\ t\varphi(t) \\ \text{for every } c \end{array}$$

	<div>1</div> $\begin{array}{c} t(\neg\sigma) \\ \\ f\sigma \end{array}$	<div>2</div> $\begin{array}{c} f(\neg\sigma) \\ \\ t\sigma \end{array}$	
<div>3</div> $\begin{array}{c} t(\sigma_1 \vee \sigma_2) \\ / \quad \backslash \\ t\sigma_1 \quad t\sigma_2 \end{array}$	<div>4</div> $\begin{array}{c} f(\sigma_1 \vee \sigma_2) \\ \\ f\sigma_1 \\ \\ f\sigma_2 \end{array}$	<div>5</div> $\begin{array}{c} t(\sigma_1 \wedge \sigma_2) \\ \\ t\sigma_1 \\ \\ t\sigma_2 \end{array}$	<div>6</div> $\begin{array}{c} f(\sigma_1 \wedge \sigma_2) \\ / \quad \backslash \\ f\sigma_1 \quad f\sigma_2 \end{array}$
<div>7</div> $\begin{array}{c} t(\sigma_1 \rightarrow \sigma_2) \\ / \quad \backslash \\ f\sigma_1 \quad t\sigma_2 \end{array}$	<div>8</div> $\begin{array}{c} f(\sigma_1 \rightarrow \sigma_2) \\ \\ t\sigma_1 \\ \\ f\sigma_2 \end{array}$	<div>9</div> $\begin{array}{c} t(\sigma_1 \leftrightarrow \sigma_2) \\ / \quad \backslash \\ t\sigma_1 \quad f\sigma_1 \\ \quad \quad \\ t\sigma_2 \quad f\sigma_2 \end{array}$	<div>10</div> $\begin{array}{c} f(\sigma_1 \leftrightarrow \sigma_2) \\ / \quad \backslash \\ t\sigma_1 \quad f\sigma_1 \\ \quad \quad \\ f\sigma_2 \quad t\sigma_2 \end{array}$
<div>11</div> $\begin{array}{c} t(\forall x)\phi(x) \\ \\ t\phi(c) \\ \text{for all } c \end{array}$	<div>12</div> $\begin{array}{c} f(\forall x)\phi(x) \\ \\ f\phi(c) \\ \text{for new } c \end{array}$	<div>13</div> $\begin{array}{c} t(\exists x)\phi(x) \\ \\ t\phi(c) \\ \text{for new } c \end{array}$	<div>14</div> $\begin{array}{c} f(\exists x)\phi(x) \\ \\ f\phi(c) \\ \text{for all } c \end{array}$

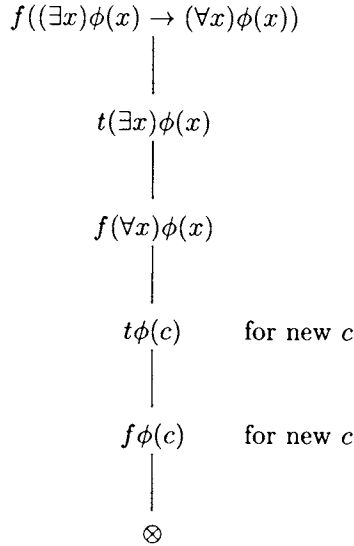
we represent the fact “for $(\forall x) \varphi(x)$ to be true, $\varphi(x)$ has to be true for every constant c ”.

Example 2.8.3:

2.8.3

The semantic tableau of $t(\forall x) \varphi(x)$ -dually, of $f(\exists x) \varphi(x)$ - allows us to declare $\varphi(c)$ true -dually, false- for every c . The semantic tableau $t(\exists x) \varphi(x)$ allows us to declare $\varphi(c)$ true only for those constants c which have not previously occurred in the systematic tableau.

The following example demonstrates what would happen without this limitation.

Example 2.8.4: Let $(\exists x)\varphi(x) \rightarrow (\forall x)\varphi(x)$ be a sentence. This sentence is not logically true since the existence of an x , such that $\varphi(x)$ holds, does not imply that $\varphi(x)$ holds for every x (for example, the existence of an $x > 3$ does not imply that for every x , $x > 3$ holds). But:



In node 5 we did not have the right to use the same constant c as in the previous node 4. We have thus “proved” that $(\exists x)\varphi(x) \rightarrow (\forall x)\varphi(x)$ is a logically true sentence, whereas it is obviously not. ■ 2.8.4

Due to the tableaux 11 and 14, a systematic tableau may continue infinitely if there is no contradiction occurring in one of its branches. (In Examples 2.8.2 and 2.8.3, there was no need to write down all the constants of the tableaux 4 and 13). This fact will surely be better understood in the following formal construction of a complete systematic tableau of a sentence φ .

Construction 2.8.5: Construction of a Complete Systematic Tableau:

The construction begins with the signed formula $f\varphi$ or $t\varphi$ as the origin of the tableau. We then proceed inductively.

Step n :

We have already formed a tableau T_n .

T_n will be extended to a new tableau T_{n+1} , by using some of the nodes of T_n .

Step $n + 1$:

Let X be the unused and non-atomic node which is furthest left from those nodes equidistant from the origin. If there is no such node X , the systematic tableau is complete. If now there is such a node, we construct tableau T_{n+1} by extending every non-contradictory branch passing through X with the concatenation (at the end of the branch) of the tableau corresponding to X . We analyse the new cases:

Case 1: X is $t((\forall x)\varphi(x))$

Let c_n be the first constant symbol of the list of all the constants of the language, such that c_n does not occur in any branch passing through X . We then add $t\varphi(c_n)$ at the end of every non contradictory branch passing through X , according to the tableau:

$$\begin{array}{c} t((\forall x)\varphi(x)) \\ | \\ t\varphi(c_n) \end{array}$$

Case 2: X is $f((\forall x)\varphi(x))$

Let c_k be the first constant symbol of the list which does not occur in any node of the branch passing through X . Then we add $f\varphi(c_k)$ at the end of every branch passing through X , according to the tableau:

$$\begin{array}{c} f(\forall x)\varphi(x) \\ | \\ f\varphi(c_k) \end{array}$$

Cases 3, 4: X is $f((\exists x)\varphi(x))$ and $t((\exists x)\varphi(x))$ respectively.

These cases are dual to 1 and 2.

Intuitively, in cases 1 and 3 (and dually in 2 and 4) we wish to avoid the repetition and we declare $\varphi(c)$ true for continually new constants, thus using up the list of constants (not in a finite period of time of course). ■ 2.8.5

Definition 2.8.6: A **Complete Systematic Tableau**, CST for short, is the union of all the tableaux T_n of the previous construction, i.e.:

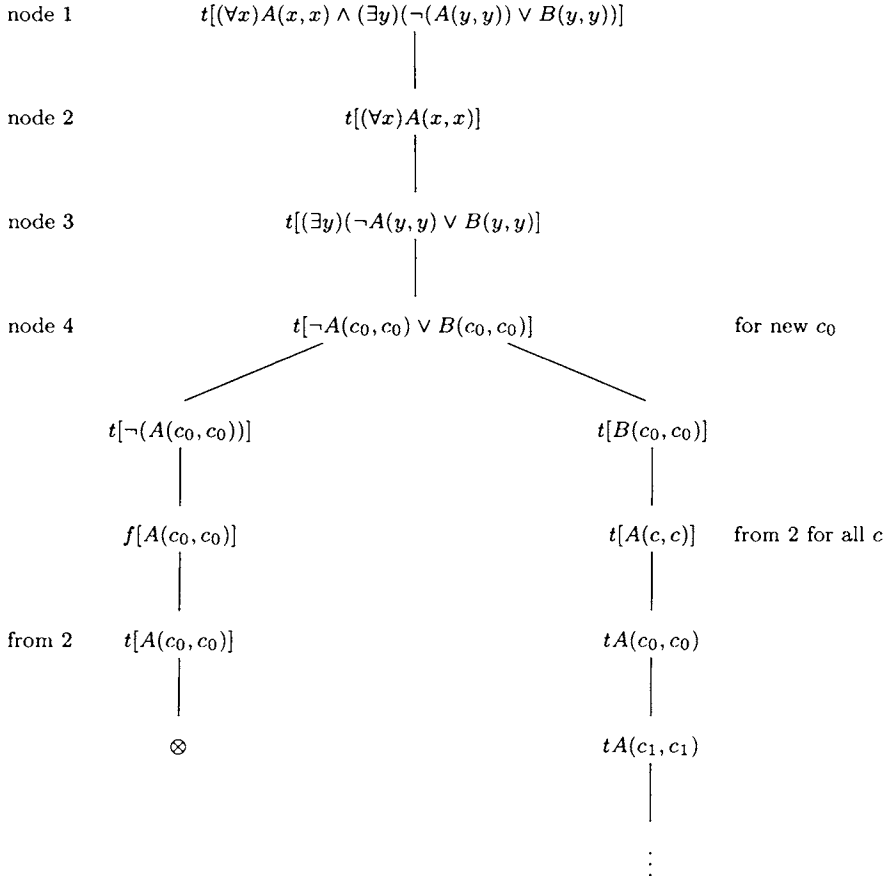
$$T = \cup_{n \in \mathbb{N}} T_n \quad \blacksquare \quad 2.8.6$$

A CST can have an infinite number of nodes, whereas the semantic tableaux of PL are always finite.

Definition 2.8.7:

- (i) A CST is **contradictory** if all its branches are contradictory.
- (ii) A sentence σ is **Beth-provable** (**Beth-refutable**) if there exists a contradictory CST with an origin $f\sigma(t\sigma)$. The fact that σ is Beth-provable is denoted by $\vdash_B \sigma$.
- (iii) A sentence σ is **Beth-provable** from a set of PrL sentences S if there exists a contradictory CST with an origin $f\sigma$ and a next node tP , where P is the conjunction of the sentences of S . This is denoted by $S \vdash_B \sigma$. ■ 2.8.7

Example 2.8.8:



In this example, the left branch is contradictory whereas the right branch continues infinitely. ■ 2.8.8

Proofs by means of semantic trees are quite similar to Beth-proofs. Let us look at an informal description of the problem and the method.

Semantic trees: Informal description

Let φ be a sentence and S the corresponding set of clauses of φ . If φ is non-satisfiable, then it is true in a Herbrand interpretation (Theorem 2.7.7). Accordingly, all the ground instances of the clauses of S are true in this interpretation. Hence, if φ is non-satisfiable, then every attempt to verify all the ground instances of the clauses of S , by means of a valuation of the ground atoms $R(t_1, t_2, \dots, t_n)$, where t_1, t_2, \dots, t_n belong to the Herbrand universe, has to fail. This failure is confirmed in a finite number of steps by the construction of a finite set of non-satisfiable ground instances in a Herbrand interpretation. So by Theorem 2.7.7, these instances are non-satisfiable in all interpretations.

Accordingly, the question is how to construct a procedure which, starting with a sentence φ and the corresponding set of clauses S ,

- (i) if φ is non-satisfiable, ends after a finite number of steps in a finite set of ground instances.
- (ii) if φ is satisfiable, the procedure does not result in anything in a finite period of time, however it constitutes the construction of a Herbrand interpretation satisfying φ .

In other words, by means of this procedure we want to obtain the proof or a counterexample of the non-satisfiability of the sentence. The construction of such a procedure requires the use of semantic trees [ChLe73, Dela87].

Definition 2.8.9: A **tree** is a structure $T = \{X, r\}$ where X is the set of the nodes of T and r is a binary relation in X such that:

- (1) if $x, y \in X$ and xry , then x is called the **previous node** of y and y the **next node** of x .
- (2) There is exactly one node of T which does not have a previous node. This node is called the **origin** of T .
- (3) Each node which differs from the origin of T has exactly one previous node.

The line connecting a node and its next node is called an **arc** of T . A **final node** is a node with no next nodes. The arc connecting a final node and the origin of T is called a **branch** of T . ■ 2.8.9

Example 2.8.10:

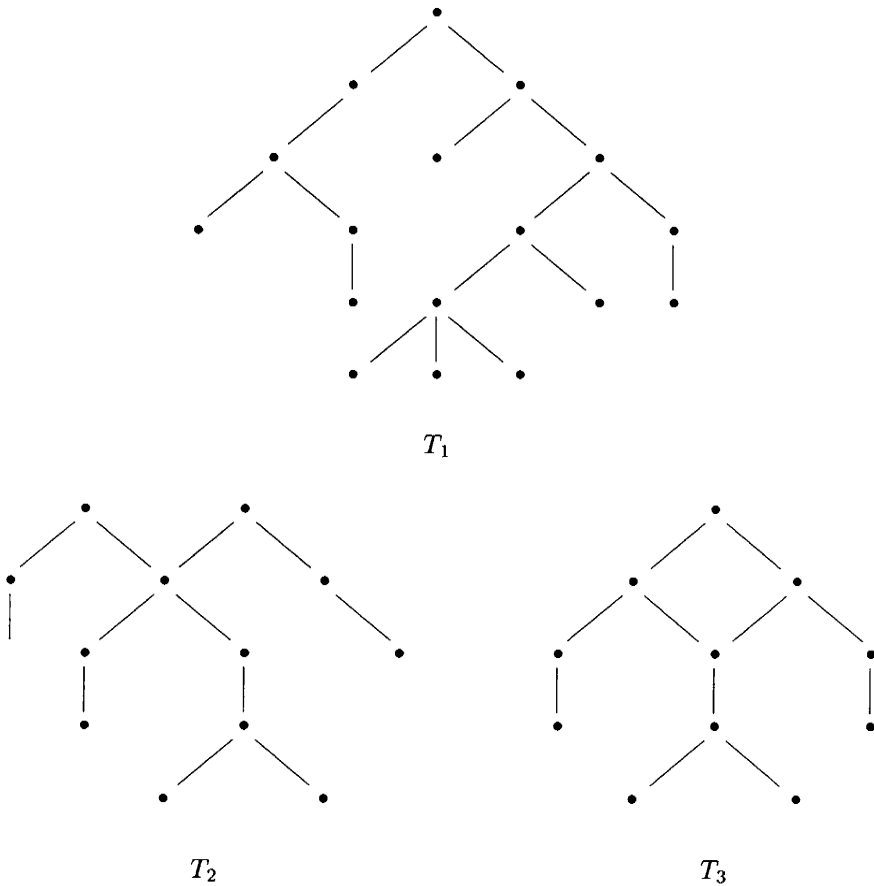


Diagram T_1 is a tree. Note the branching on the lower part which resembles the foliage of a tree turned upside down. T_2 is *not* a tree, it has two origins and, furthermore, there is one node with two previous nodes. T_3 is also not a tree, it has a node with two previous nodes. ■ 2.8.10

Remark 2.8.11: The semantic tableaux of a PL proposition and the complete systematic tableaux of a PrL sentence are trees. ■ 2.8.11

Definition 2.8.12: Semantic Trees:

Let S be a set of clauses, $S = \{C_1, \dots, C_n\}$, P_1, \dots, P_λ the atoms occurring in the clauses of S , and $\{a_1, \dots, a_n\}$ the Herbrand universe of S . A **semantic tree** for S is a tree T such that:

- (1) The origin of T is an arbitrary point. The nodes differing from the origin are ground instances of P_1, \dots, P_λ in the universe $\{a_1, \dots, a_n, \dots\}$. Each node has exactly two next nodes, i.e. $P_i(a_{i_1}, \dots, a_{i_i})$ and $\neg P_i(a_{i_1}, \dots, a_{i_i})$.
- (2) Every branch of T containing exactly the ground instances

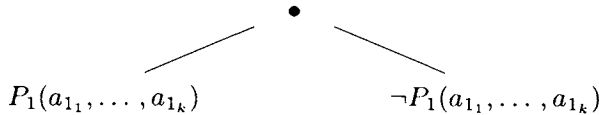
$$P_{i_1}(a_{i_1}, \dots, a_{i_k}), \dots, P_{i_p}(a_{p_1}, \dots, a_{p_k})$$

represents the conjunction

$$P_{i_1}(a_{i_1}, \dots, a_{i_k}) \wedge \dots \wedge P_{i_p}(a_{p_1}, \dots, a_{p_k})$$

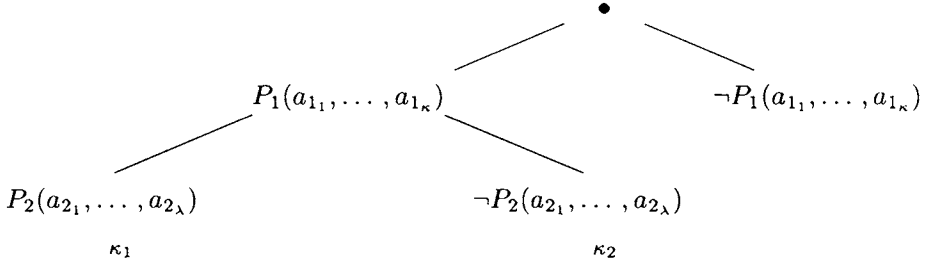
- (3) The disjunction of all the conjunctions of the branches of T is logically true.
- (4) If a node of T is $P_i\{a_{i_1}, \dots, a_{i_i}\}$ then the next node cannot be $\neg P_i(a_{i_1}, \dots, a_{i_i})$.
- (5) If during the construction of T we are at a node k which contradicts a ground instance of one of C_1, \dots, C_n of S , then k is a final node and the corresponding branch is said to be **contradictory**. ■ 2.8.12

In practice, in order to construct the semantic tree of S , we start with $P_1(a_{i_1}, \dots, a_{i_k})$:



If one of the clauses of S contains $P_1\{a_{1_1}, \dots, a_{1_k}\}$, then the branch on the right is contradictory, $\neg P_1(a_{1_1}, \dots, a_{1_k})$ is a final node and the construction continues with the left branch. If one of the clauses of S contains $\neg P_1(a_{1_1}, \dots, a_{1_k})$, then the branch on the left is contradictory, $P_1(a_{1_1}, \dots, a_{1_k})$ is a final node and the construction continues with the right branch.

Let us assume that $P_1(a_{1_1}, \dots, a_{1_k})$ is not a final node. We continue with the node $P_2(a_{2_1}, \dots, a_{2_\lambda})$. (The choice of the atom with which we continue the construction is ours).



Branch κ_1 is the conjunction $P_1(a_{1_1}, \dots, a_{1_k}) \wedge P_2(a_{2_1}, \dots, a_{2_\lambda})$. Branch κ_2 is the conjunction $P_1(a_{1_1}, \dots, a_{1_k}) \wedge \neg P_2(a_{2_1}, \dots, a_{2_\lambda})$. We check whether one of the clauses of S contains $\neg P_1(a_{1_1}, \dots, a_{1_k})$ or $\neg P_2(a_{2_1}, \dots, a_{2_\lambda})$. If that is the case, κ_1 is a contradictory branch and we continue with κ_2 . We continue the inspection and the construction. Our goal is to reach final nodes, using up all the atoms of S .

Definition 2.8.13:

- (1) A set of clauses of S is said to be **refutable by semantic tree** if there exists a semantic tree for S , the branches of which are contradictory.
- (2) A branch κ of a semantic tree of a set S of clauses is called **complete**, if for every ground instance $P(a_1, \dots, a_n)$ of each atom P , either $P(a_1, \dots, a_n)$ or $\neg P(a_1, \dots, a_n)$ is contained in κ . ■ 2.8.13

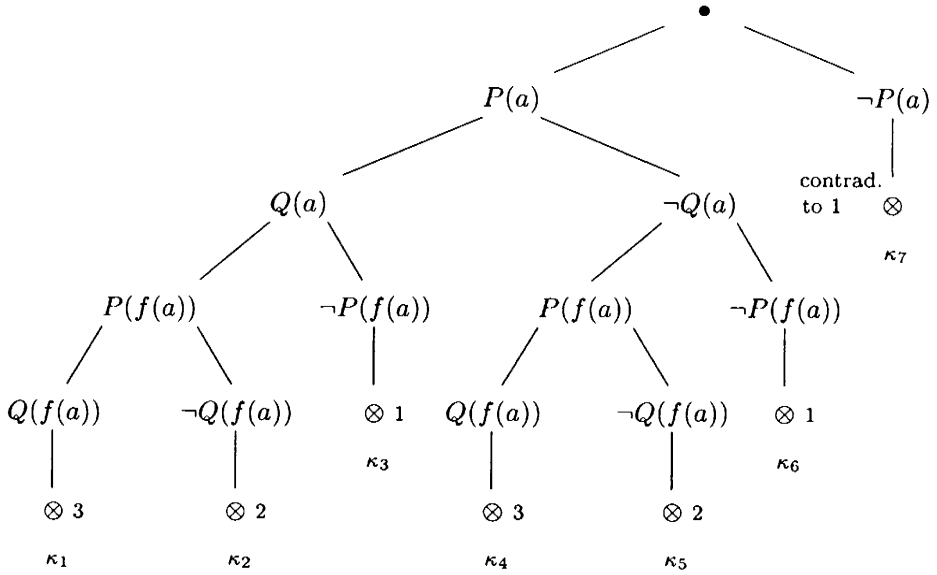
Example 2.8.14:

Assume $\varphi : (\forall x) [P(x) \wedge (\neg P(x) \vee Q(f(x))) \wedge \neg Q(f(x))]$

Then $S = \left\{ \underbrace{\{P(x)\}}_1, \underbrace{\{\neg P(x), Q(f(x))\}}_2, \underbrace{\{\neg Q(f(x))\}}_3 \right\},$

and $H = \text{Herbrand universe} = \{a, f(a), f(f(a)), f(f(f(a))), \dots\},$
 Ground instances $= \{P(a), Q(a), P(f(a)), Q(f(a)), \dots\}$

A semantic tree for S is $T_1 :$



Let us see why κ_2 is contradictory: κ_2 represents the conjunction:

$$\neg Q(f(a)) \wedge P(f(a)) \wedge Q(a) \wedge P(a)$$

The second clause of S is $\{\neg P(x), Q(f(x))\}$. By S , we demand:

$$\neg P(x) \vee Q(f(x))$$

to be valid for every x . However κ_2 states that there is an x in the Herbrand universe which is actually equal to a and for which we have:

$$\neg Q(f(a)) \wedge P(f(a)) \wedge Q(a) \wedge P(a)$$

Then we also have:

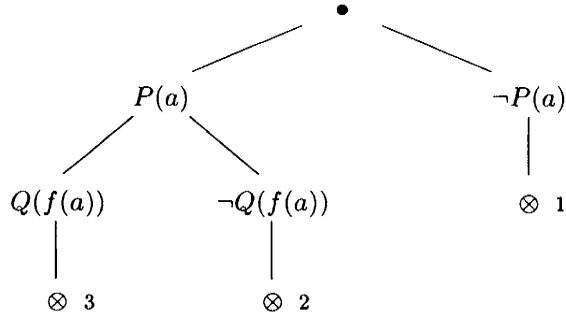
$$\neg Q(f(a)) \wedge P(a) \quad (\text{by } A \wedge B \rightarrow A),$$

and by De Morgan:

$$\neg(Q(f(a)) \vee \neg P(a))$$

in other words κ_2 gives us contradiction with the second clause of S .

As we have said before, we choose the order in which we assign values from H to the atoms of S . Thus, if we choose $Q(f(a))$ after $P(a)$, we have a very simple semantic tree for S , T_2 , which has all of its branches contradictory:



■ 2.8.14

Example 2.8.15:

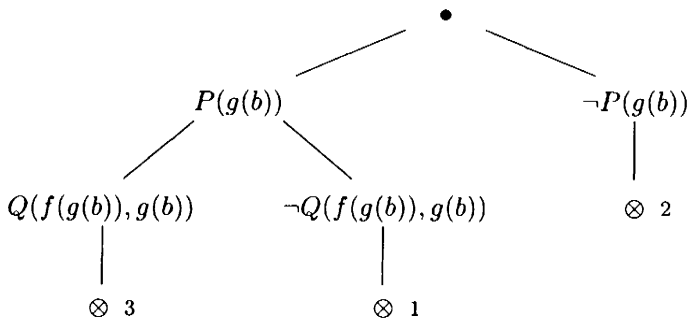
Assume $\varphi : (\forall x) (\forall z) ((\neg P(x) \vee Q(f(x), x)) \wedge P(g(b)) \wedge \neg Q(x, z))$

Then $S = \{\underbrace{\{\neg P(x), Q(f(x), x)\}}_1, \underbrace{\{P(g(b))\}}_2, \underbrace{\{\neg Q(x, z)\}}_3\},$

and $H = \{b, f(b), g(b), f(f(b)), f(g(b)), \dots\}$

Ground instances = $\{P(b), Q(f(b), b), P(g(b)), Q(f(g(b)), g(b)), \dots\}$

Semantic tree for S :



■ 2.8.15

As said before, the order in which we use the atoms in the construction of the tree has an important effect on the number of steps required to reach final nodes and contradictory branches.

In Remark 2.8.11, we saw that the semantic tableaux and the complete systematic tableaux are trees. By Koenig's lemma, Lemma 1.11.9, every finite tree with infinite nodes has at least one infinite branch. Then, if S is not satisfiable, the construction of its semantic tree will lead through a finite number of steps to the finding of a Herbrand interpretation H , such that $A_H \not\models S$. If S is satisfiable, then the corresponding construction will result in an infinite tree, every branch of which will determine a Herbrand interpretation satisfying S . The above conclusion constitutes the substance of Herbrand's theorem, which we will next prove by using a method for the construction of a semantic tree corresponding to a given set S of clauses.

Theorem 2.8.16: Theorem of Herbrand:

If S is a non-satisfiable set of clauses, then S is refutable by a semantic tree.

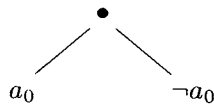
Proof: We will describe an algorithm for the construction of a semantic tree for S . We form the Herbrand Universe for S as well as the set

$$\{a_0, a_1, \dots\}$$

of the ground instances of the atoms of S . Then

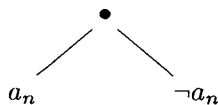
Step 0:

We construct the tree:



Step n:

We concatenate, at the final node of every non-contradictory branch κ , the extension:



Let us now assume that S is not refutable by a semantic tree. Then the construction described by the above algorithm will never end. However in that case, Koenig's lemma guarantees that the tree will contain an infinite branch κ . For every ground instance a_n , κ contains either a_n or its negation $\neg a_n$ as the name of the node.

We now define a Herbrand interpretation as follows:

For every n -ary predicate symbol P , and terms t_1, t_2, \dots, t_n , with their interpretation belonging to the Herbrand universe of S , the interpretation of P is the relation:

$$\varepsilon(P) (\varepsilon(t_1), \dots, \varepsilon(t_n))$$

where $P(t_1, \dots, t_n)$ is the name of some node of the infinite branch. The interpretation obviously satisfies all the clauses of S . Then S is satisfiable. ■ 2.8.16

In fact, the theorem of Herbrand provides an algorithm examining the satisfiability of a sentence or a set of clauses S with propositional logic methods, for example tableaux or resolution. If S is not satisfiable, then there is a set of ground instances of the clauses of S which is not satisfiable. This finite set consists of PL propositions, and its non-satisfiability can be evidenced with methods which are known to us. Thus, for every set of clauses S , we start counting all the ground instances of the clauses of S . While this procedure is under way, we systematically check the satisfiability of every finite subset of ground instances with propositional logic methods. If S is not satisfiable, then this inspection will show that one of the finite subsets is not satisfiable. If S is satisfiable, the inspection will continue infinitely.

Example 2.8.17: Assume the sentence of Example 2.8.15.

$$\varphi : (\forall x) (\forall z) ((\neg P(x) \vee Q(f(x), x)) \wedge P(g(b)) \wedge \neg Q(x, z))$$

Determine whether φ is satisfiable. The corresponding set of clauses is:

$$S = \underbrace{\{\neg P(x), Q(f(x), x)\}}_1, \underbrace{\{P(g(b))\}}_2, \underbrace{\{\neg Q(y, z)\}}_3$$

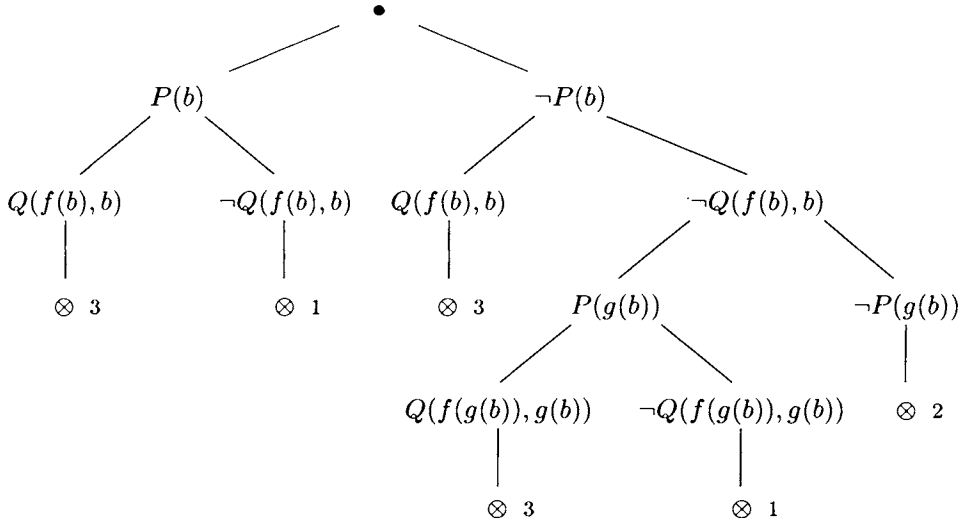
The Herbrand Universe for S is:

$$H = \{b, g(b), f(b), f(g(b)), g(f(b)), \dots\}.$$

The set of ground instances of the atoms of S is

$$\{P(b), Q(f(b), b), P(g(b)), Q(f(g(b)), g(b)), P(f(b)), \dots\}$$

We construct the systematic semantic tree for S according to the method of Theorem 2.8.16.



This semantic tree constitutes the proof that S is not satisfiable and gives us a certain subset of ground instances of clauses of S which is not satisfiable. These ground instances are those, and only those, which were used in order to proclaim a certain branch contradictory. Concretely :

$$\left\{ \{ \neg Q(f(b), b) \}, \{ \neg Q(f(g(b)), g(b)) \}, \{ P(g(b)) \}, \right. \\ \left. \{ \neg P(b), Q(f(b), b) \}, \{ \neg P(g(b), Q(f(g(b)), g(b)) \} \right\}$$

The two first instances come from clause 3 of S , the third from 2 and the two last from 1. The non-satisfiability of this finite set is indeed proved by the method of resolution in the context of Propositional Logic. Let us name:

$$\begin{aligned} A : & Q(f(b), b) \\ B : & Q(f(g(b)), g(b)) \\ C : & P(g(b)) \\ D : & P(b) \end{aligned}$$

We can then write this finite subset of the ground instances of the clauses of S as follows:

$$S' = \underbrace{\{\neg A\}}_1, \underbrace{\{\neg B\}}_2, \underbrace{\{C\}}_3, \underbrace{\{\neg D, A\}}_4, \underbrace{\{\neg C, B\}}_5$$

Using resolution we have:

- (1) $\neg A$
- (2) $\neg B$
- (3) C
- (4) $\neg D, A$
- (5) $\neg C, B$
- (6) B by (3) and (5)
- (7) \square by (2) and (6)

We must at this point note that the algorithm lying in the construction of a systematic semantic tree does not always yield the minimal non-satisfiable set of ground instances.

Thus in the previous example

$$S_1 = \underbrace{\{\{\neg B\}\}}_1, \underbrace{\{C\}}_2, \underbrace{\{\neg C, B\}}_3$$

is already a non-satisfiable subset.

- (1) $\neg B$
- (2) C
- (3) $\neg C, B$
- (4) B by (2) and (3)
- (5) \square by (1) and (4) ■ 2.8.17

Example 2.8.18: Let us look at a satisfiable set of clauses. Assume sentence

$$\varphi : (\forall x) [(\exists y) P(x, y) \rightarrow (\exists y) P(a, y)].$$

The SNF of φ is $(\forall x) (\forall y) [\neg P(x, y) \vee P(a, f(x, y))]$. (Why?)

The corresponding set of clauses of φ is:

$$S = \{\{\neg P(x, y), P[a, f(x, y)]\}\}$$

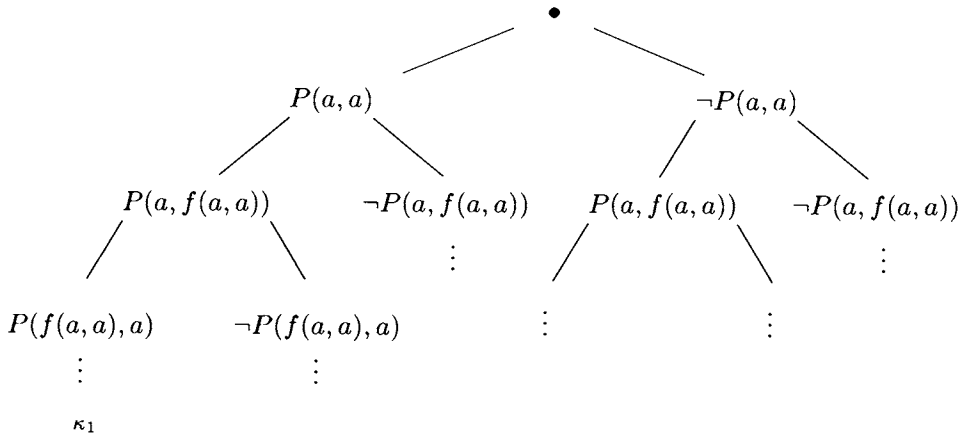
and the corresponding Herbrand universe:

$$H = \{a, f(a, a), f(a, f(a, a)), f(f(a, a), a), f(f(a, a), f(a, a)), \dots\}$$

The ground instances of the atoms of S are:

$$\{P(a, a), P(a, f(a, a)), P(f(a, a), a), P(f(a, a), f(a, a)), \dots\}$$

A semantic tree for S is:



This infinite systematic tree contains branches which are not contradictory by their construction, since the conjunction of their atoms does not go against any clauses of S . For example, branch κ_1 which does not contain negations of P is non-contradictory. κ_1 gives us a Herbrand interpretation of φ satisfying S .

■ 2.8.18

2.9 Unification and Resolution in PrL

We have already referred to the Prenex and Skolem Normal forms of a sentence. Let φ be a sentence of a PrL language. Then, by following the steps

- A: Prenex Form
- B: CNF of the subsentence containing no quantifiers
- C: Skolem Normal Form
- D: Clausal Form

we can reduce φ to a set of clauses and determine its truth value with the usual PrL methods.

Example 2.9.1: Let φ be a sentence which is already in a Prenex Form (A).

$$\begin{aligned}\varphi : & (\forall x) (\exists y) (\exists z) ((\neg P(x, y) \wedge Q(x, z)) \vee R(x, y, z)) \\ & \leftrightarrow (\forall x) (\exists y) (\exists z) ((\neg P(x, y) \vee R(x, y, z)) \wedge (Q(x, z) \vee R(x, y, z)))\end{aligned}$$

We now introduce the Skolem function symbols f, g , where $y = f(x)$ and $z = g(x)$. Then

$$\varphi \leftrightarrow (\forall x) [(\neg P(x, f(x)) \vee R(x, f(x), g(x))) \wedge (Q(x, g(x)) \vee R(x, f(x), g(x)))]$$

and we finally determine the clausal form (D) of:

$$S = \{ \{ \neg P(x, f(x)), R(x, f(x), g(x)) \}, \{ Q(x, g(x)), R(x, f(x), g(x)) \} \}$$

■ 2.9.1

In the clausal form of φ , we have to deal with instantiation of variables and with Skolem functions. The semantic trees defined in the previous section help us deal with such problems. Using semantic trees, we can select symbols for the substitution of variables from the corresponding Herbrand universe and apply the method of resolution in order to find contradictions among the ground instances of the occurring clauses. Such a procedure is, however, time consuming. What's more, it cannot easily be used as a structured conclusion mechanism likely to be programmed into a computer. We therefore need a more algorithmic method which can be used with a computer. The unification procedure which will now be examined [ChLe73, Dela87, Fitt90, Lloy87, Thay88] offers a method such as the one required for the finding of contradictions.

Unification: Informal Description

We have already defined, Definition 2.2.19, the concept of substitution. Assume we are given the following clauses

$$C_1 : \{ P(f(x), y), Q(a, b, x) \} \quad \text{and} \quad C_2 : \{ \neg P(f(g(c)), g(d)) \}$$

We want to apply resolution to C_1 and C_2 substituting x by $g(c)$ and y by $g(d)$. For this we have to

- (1) check whether C_1 and C_2 can be resolved, and
- (2) find the suitable substitution sets allowing resolution.

These controls and substitutions are achieved with the unification algorithm. Let us now see how this algorithm unifies C_1 and C_2 .

Step 1 :

We start by comparing the clause's terms on the left side and proceed towards the right side until we meet the first terms with the same function symbol which do not agree in the variables or in the constants. We now create a set containing those terms, the **disagreement set**. For C_1 and C_2 , $\{x, g(c)\}$ is the first disagreement set.

Step 2 :

We check each variable of the disagreement set to see if it occurs in some other term of that same set.

Step 3 :

If the previous check is positive, then the clauses do not unify and the algorithm ends with a failure. If it is negative, we proceed with the substitution of the variable by the other term of the disagreement set. For C_1 and C_2 we apply substitution $\theta_1 = \{x/g(c)\}$. C_1 and C_2 then become:

$$\begin{aligned} C_1^1 &= \{P(f(g(c)), y), Q(a, b, x)\} \\ C_2^1 &= C_2 \end{aligned}$$

Step 4 :

We proceed to the right by following steps 1 – 3. The new disagreement set is $\{y, g(d)\}$. We apply substitution $\theta_2 = \{y/g(d)\}$. We impose:

$$\begin{aligned} C_1^2 &= \{P(f(g(c)), g(d)), Q(a, b, x)\} \\ C_2^2 &= C_2 \end{aligned}$$

Resolution can obviously be applied to C_1^2 and C_2^2 .

Finally the algorithm will finish by giving a set of substitutions by means of which C_1 and C_2 are unified and then resolved by the PL resolution rule. This set of substitutions is called a **general unifier**, GU, of the resolved clauses. For C_1 and C_2 , the general unifier is $\theta = \{x/g(c), y/g(d)\}$. If the clauses cannot be resolved, the algorithm will finish in step 2.

Unification: Formal Description

We will now continue with the formal description of the unification algorithm and the necessary definitions.

Definition 2.9.2: Disagreement Set:

Let $S = \{C_1, C_2, \dots, C_n\}$ be a set of clauses. The set:

$$\text{DS}(S) = \{t_i, t_{j_1}, \dots, t_{j_\lambda} \mid t_i \text{ is the first term on the left which occurs in some subterm } c_k \text{ of the clauses of } S \text{ and terms } t_{j_1}, \dots, t_{j_\lambda} \text{ occur in subterms } c_{j_1}, \dots, c_{j_\lambda} \text{ of the clauses of } S, \text{ so that for substitution } \theta_1 = \{t_i/t_{j_\mu}\} \text{ or substitution } \theta_2 = \{t_{j_\mu}/t_i\}, 1 \leq \mu \leq \lambda, c_k\theta_1 \text{ is identical to } c_{j_\mu}\theta_1, \text{ or } c_k\theta_2 \text{ is identical to } c_{j_\mu}\theta_2\}$$

is called the **disagreement set** of S , DS for short. ■ 2.9.2

The disagreement set is not uniquely defined, it depends on the order of inscription of the clauses of S .

In Example 2.9.1 we have

$$S = \{C_1, C_2\} = \{\{P(f(x), y), Q(a, b, x)\}, \{\neg P(f(g(c)), g(d))\}\}$$

If we substitute $g(c)$ for the subterm x of $f(x)$ in $P(f(x), y)$, in other words $\theta_1 = \{x/g(c)\}$, then $f(x)\theta_1$ is identical to $f(g(c))\theta_1$. The first disagreement set is thus $\text{DS}_1 = \{x, g(c)\}$. After the application of θ_1 the disagreement between x and $g(c)$ is raised and the next disagreement located by the algorithm is given by the set $\text{DS}_2 = \{y, g(c)\}$.

Definition 2.9.3: Let $X = \{\sigma_1, \sigma_2 \dots \sigma_n\}$ be a set of clauses or terms. A substitution θ is called a **unifier** for C if $\sigma_1\theta = \sigma_2\theta = \dots = \sigma_n\theta$. ■ 2.9.3

Definition 2.9.4: A unifier θ is called a **Most General Unifier**, MGU for short, if for every other unifier ψ there is a substitution γ such that $\psi = \theta\gamma$. ■ 2.9.4

For a set S of terms or clauses, the Most General Unifier is uniquely determined [Lloy87].

Example 2.9.5: Assume the set of atoms $S = \{Q(g(x), w), Q(y, b)\}$, where b is a constant symbol. A unifier of S is:

$$\psi = \{y/g(b), x/b, w/b\}$$

If ψ is applied to S , it creates the following ground instances:

$$\begin{aligned} Q(g(x), w) \psi &= Q(g(b), b) \\ Q(y, b) \psi &= Q(g'(b), b) \end{aligned}$$

If we now impose $\gamma = \{x/b\}$ and $\theta = \{y/g(x), w/b\}$ we can easily prove that $\psi = \theta \gamma$.

Furthermore, θ unifies the above atoms. θ is thus an MGU, most general unifier of S . ■ 2.9.5

Example 2.9.6: Let C be the set of terms $C = \{f(x, g(x)), f(h(y), g(h(y)))\}$.

$\psi = \{x/h(g(c)), y/g(c)\}$ is a unifier of C . $\theta = \{x/h(y)\}$ is also a unifier of C . If we impose $\gamma = \{y/c\}$, then $\psi = \theta \gamma$ can easily be proved. Hence θ is a general unifier of C . ■ 2.9.6

We can now move to the formal description of the unification algorithm.

Algorithm 2.9.7 Unification algorithm:

Let $T = \{P_1, P_2, \dots, P_n\}$ be a set of atomic formulae.

Step 0:

$\theta_0 = E$ (identical substitution)

Step k:

We already have substitutions $\theta_0, \theta_1, \dots, \theta_k$

Step k+1: (recursive step):

- (i) If $P_1\theta_1\theta_2\dots\theta_k = P_2\theta_1\theta_2\dots\theta_k = \dots = P_n\theta_1\theta_2\dots\theta_k$ then the algorithm finishes by providing $\theta = \theta_1\theta_2\dots\theta_k$ as a general unifier.
- (ii) If $P_i\theta_1\theta_2\dots\theta_k \neq P_j\theta_1\theta_2\dots\theta_k$ for some i, j , then:

- (a) We form the disagreement set

$$DS(P_1\theta_1\dots\theta_k, \dots, P_n\theta_1\dots\theta_k) = DS(T\theta_1\dots\theta_k) = DS_k$$

- (b) We carry out the **Occur Check** (OC) of the variables, in other words we check whether every variable $v \in DS_k$ occurs in some other element of DS_k .

If the OC is affirmative, then we stop and conclude that T is not unifiable. If there is no variable which belongs to DS_k then the OC is regarded as affirmative. If there are $v, t \in DS_k$ and the OC is negative then we impose $\theta_{k+1} = \{v/t\}$ to eliminate the disagreement between P_1, \dots, P_n in terms v, t .

Step $k+2$:

We resume steps $k+1, k+2$ for $k = k+1$.

As we will see clearly with the next theorem, if T is unifiable, the algorithm always finishes by providing the most general unifier:

$$MGU = \theta = \theta_1 \theta_2 \dots \theta_n \quad \blacksquare \quad 2.9.7$$

The proof of the following theorem is beyond the scope of this book, it can be found in [Robi65, ChLe73].

Theorem 2.9.8: (J. A. Robinson)

If we apply the unification algorithm to $T = \{P_1, P_2, \dots, P_n\}$, then:

If T is unifiable then the algorithm finishes by providing the MGU of T .

If T is not unifiable then the algorithm finishes by declaring that there is no unifier. $\blacksquare \quad 2.9.8$

The above theorem states that if T is unifiable, then the unification algorithm finishes by determining the MGU (and not only some unifier) of T .

Example 2.9.9: Assume the set of formulae:

$$S = \{Q(a, x, f(g(z))), Q(z, f(y), f(y))\}$$

Is S unifiable? If it is, determine the MGU.

Step 0: We impose $\theta_0 = E$

Step 1: $S\theta_0 = S$

$DS(S\theta_0) = DS_1 = \{a, z\}$

OC negative

We impose $\theta_1 = \{z/a\}$

Step 2: $S\theta_0\theta_1 = \{Q(a, x, f(g(a))), Q(a, f(y), f(y))\}$
 $DS(S\theta_0\theta_1) = DS_2 = \{x, f(y)\}$
 OC negative
 We impose $\theta_2 = \{x, f(y)\}$

Step 3: $S\theta_0\theta_1\theta_2 = \{Q(a, f(g(a)), f(g(a))), Q(a, f(g(a)), f(g(a)))\}$
 $DS(S\theta_0\theta_1\theta_2) = DS_3 = \{g(a), y\}$
 OC negative
 We impose $\theta_3 = \{y/g(a)\}$

Step 4: $S\theta_0\theta_1\theta_2\theta_3 = \{Q(a, f(a)), f(g(a))), Q(a, f(g(a)), f(g(a)))\}$
 $DS(S\theta_0\theta_1\theta_2\theta_3) = \{Q(a, f(g(a)), f(g(a)))\}$, singleton

S is hence unifiable and its MGU is:

$$MGU = \theta_0\theta_1\theta_2\theta_3 = \{z/a, x/f(g(a)), y/g(a)\} \quad \blacksquare \quad 2.9.9$$

Example 2.9.10: Assume the set of formulae:

$$S = \{Q(y, y), Q(z, f(z))\}.$$

Is S unifiable? If it is, determine a GU.

Step 0: $\theta_0 = E$

Step 1: $S\theta_0 = S$
 $DS(S\theta_0) = DS_1 = \{y, z\}$
 OC negative
 We impose $\theta_1 = \{y/z\}$

Step 2: $S\theta_0\theta_1 = \{Q(z, z), Q(z, f(z))\}$
 $DS(S\theta_0\theta_1) = DS_2 = \{z, f(z)\}$
 OC affirmative, z occurs in $f(z)$.

Then S is not unifiable.

$\blacksquare \quad 2.9.10$

We must note at this point that in many applications, in the pursuit of higher efficiency, the PROLOG inference mechanism based on the unification algorithm ignores Occur Check. In other words, it substitutes the first term of a given DS for the first variable x . This can surely lead to erroneous conclusions; it is therefore for the programmer to create the suitable flow and security mechanisms in the program in order to avoid such errors.

We are now in position to describe the PrL resolution method in simple terms.

Resolution in PrL

The PrL resolution method is actually a combination of PrL unification and PL resolution. Thus, just as in PL (Definition 1.9.17), if S is a set of PrL clauses, then a **proof by resolution from S** , is a finite sequence of clauses C_1, \dots, C_n , such that for every C_i ($1 \leq i \leq n$), we have $C_i \in S$ and $C_i \in R(\{C_j, C_k\})$ ($1 \leq j, k \leq i$), where $R(\{C_j, C_k\})$ is the resolvent of C_j and C_k .

Note here that since the variables of all the clauses are considered bound by universal quantifiers, we can rename these variables to avoid confusions in each unification. This renaming procedure is called a **normalization of variables**.

Let us take a look at an example.

Example 2.9.11: Consider the following clauses:

$$\begin{aligned} C_1 &= \{\neg P(x, y), \neg P(y, z), P(x, z)\} \\ C_2 &= \{\neg P(u, v), P(v, u)\} \end{aligned}$$

We wish to conclude:

$$C_3 = \{\neg P(x, y), \neg P(z, y), P(x, z)\}$$

The respective notations for C_1, C_2 and C_3 in the context of PL are:

$$\begin{aligned} (\forall x) (\forall y) (\forall z) [P(x, y) \wedge P(y, z) \rightarrow P(x, z)] & \quad \text{for } C_1 \\ (\forall x) (\forall y) (\forall z) [P(u, v) \rightarrow P(v, u)] & \quad \text{for } C_2 \\ (\forall x) (\forall y) (\forall z) [P(x, y) \wedge P(z, y) \rightarrow P(x, z)] & \quad \text{for } C_3 \end{aligned}$$

Method I.

We work directly within the context of PrL:

C_1 , C_2 and C_3 are equivalent to:

$$(\forall x) (\forall y) (\forall z) [\neg P(x, y) \vee \neg P(y, z) \vee P(x, z)] \quad (1)$$

$$(\forall x) (\forall y) (\forall z) [\neg P(u, v) \vee P(v, u)] \quad (2)$$

$$(\forall x) (\forall y) (\forall z) [\neg P(x, y) \vee \neg P(z, y) \vee P(x, z)] \quad (3)$$

We rename the variables of (2):

$$(\forall x) (\forall y) (\forall z) [\neg P(x, z) \vee P(z, x)] \quad (4)$$

and by Theorem 2.3.6, (4) becomes:

$$(\forall x) (\forall y) (\forall z) [\neg P(x, z) \vee P(z, x)] \quad (5)$$

The conjunction of (1) and (5) gives, by Theorem 2.3.6:

$$(\forall x) (\forall y) (\forall z) [(\neg P(x, z) \vee P(z, x)) \wedge (\neg P(x, y) \vee \neg P(y, z) \vee P(x, z))] \quad (6)$$

and by the formula:

$$(\neg A \vee \dot{B}) \wedge (A \vee C) \rightarrow (B \vee C) \quad (*)$$

which is derivable in PrL (why?), (6) takes the form:

$$(\forall x) (\forall y) (\forall z) [P(z, x) \vee \neg P(x, y) \vee \neg P(y, z)] \quad (7)$$

We rename the variables of (2):

$$(\forall x) (\forall y) (\forall z) [\neg P(z, x) \vee P(x, z)] \quad (8)$$

and by 2.3.1, (8) becomes:

$$(\forall x) (\forall y) (\forall z) [\neg P(z, x) \vee P(x, z)] \quad (9)$$

The conjunction of (9) and (7) gives, by (*):

$$(\forall x) (\forall y) (\forall z) [\neg P(x, y) \vee \neg P(y, z) \vee P(x, z)] \quad (10)$$

We rename the variables of (2):

$$(\forall x) (\forall z) (\forall y) [\neg P(z, y) \vee P(y, z)] \quad (11)$$

By Theorem 2.3.6, (11) becomes:

$$(\forall x) (\forall z) (\forall y) [\neg P(z, y) \vee P(y, z)] \quad (12)$$

The conjunction of (10) and (12) then gives, by (*):

$$(\forall x) (\forall z) (\forall y) [\neg P(x, y) \vee \neg P(z, y) \vee P(x, z)]$$

which is the formula (3) we are seeking.

Method II.

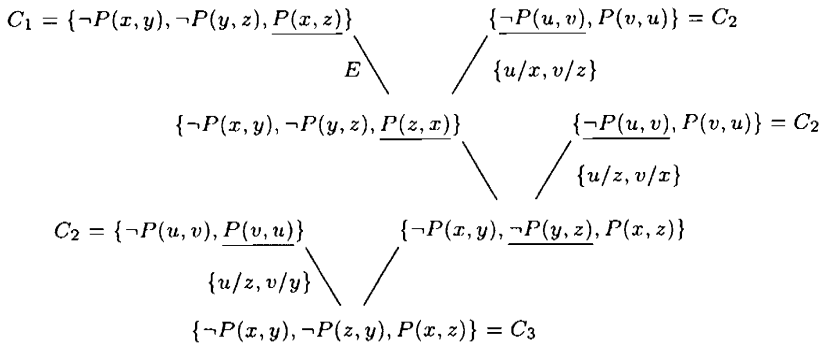
We give a proof of C_3 by resolution:

- | | | |
|-----|---|------------------------------------|
| (1) | C_1 | |
| (2) | C_2 | |
| (3) | $\{\neg P(x, z), P(z, x)\}$ | by (2), by means of $\{u/x, v/z\}$ |
| (4) | $\{\neg P(x, y), \neg P(y, z), P(z, x)\}$ | by (1) and (3), by resolution |
| (5) | $\{\neg P(z, x), P(x, z)\}$ | by (2), by means of $\{u/z, v/x\}$ |
| (6) | $\{\neg P(x, y), \neg P(y, z), P(x, z)\}$ | by (4) and (5), by resolution |
| (7) | $\{\neg P(z, y), P(y, z)\}$ | by (2), by means of $\{u/z, v/y\}$ |
| (8) | C_3 | by (6) and (7), by resolution. |

Method II offers a more mechanistic procedure which is clearly faster and more effective.

Method III.

The proof can also be given by means of a reversed tree, the origin of which is C_3 . The clauses selected for resolution occur in the same row with their variables normalized. The applied unifications occur in the branches of the tree and the corresponding atoms under resolution are underlined.



The above procedure with the tree, which we also used to prove clause C_3 , is also used by the PROLOG language. ■ 2.9.11

PROLOG's dynamism and, more generally, Logic Programming's effectiveness in the field of symbolic programming, are obvious. The third chapter gives an analytical presentation of PROLOG.

Remark 2.9.12: Working with resolution is actually an application and simplification of the corresponding PrL working method. Hence, if the sentence σ of PrL has a proof by resolution from the set S of PrL, denoted by $S \vdash_R \sigma$, then σ is provable by S , Definition 2.3.8. Formally:

$$S \vdash_R \sigma \Rightarrow S \vdash \sigma$$

The reverse is however also valid for every set S of clauses:

$$S \vdash \sigma \Rightarrow S \vdash_R \sigma \quad \blacksquare \quad 2.9.12$$

Let us now take a look at the soundness and completeness results for the methods we have presented so far.

2.10 Soundness and Completeness of PrL Proofs

We will refer here, just as in PL, to conclusions concerning the completeness and the soundness of PrL proofs. The proofs are quite similar to the PL proofs, and they can be found in [ChLe73, Meta85, Smul68].

Soundness and Completeness of Tableaux Proofs

$\vdash_B \sigma$ denotes that a sentence σ is Beth-provable, and $\models \sigma$ denotes that σ is logically true. We will give the auxiliary lemmata and theorems of the soundness and completeness of Beth-proofs.

Lemma 2.10.1: *Let R be a predicate symbol of arity n of a language \mathcal{L} , and let σ be a sentence of \mathcal{L} . Assume there is a non-contradictory branch κ in a systematic tableau with $\psi\sigma$ at the origin. We form an interpretation \mathcal{A} , the universe of which is any set $\{a_1, a_2, \dots\} = A$ which is in a one-to-one correspondence with the constant symbols of the language \mathcal{L} . The interpretation of every constant symbol c_i is the element $a_i = \varepsilon(c_i)$.*

We define a relation $\varepsilon(R) \subseteq A^n$:

$$\varepsilon(R)(a_{i_1}, a_{i_2}, \dots, a_{i_n}) \Leftrightarrow tR(c_{i_1}, c_{i_2} \dots c_{i_n}) \text{ is a node of branch } \kappa$$

Then:

- (i) if $f\sigma$ is a node of κ then σ is false in \mathcal{A} .
- (ii) if $t\sigma$ is a node of κ then σ is true in \mathcal{A} . ■ 2.10.1

Theorem 2.10.2: Completeness:

If σ is a consequence of the set of sentences S of PrL, then it is also Beth-provable by S :

$$S \models \sigma \Rightarrow S \vdash_B \sigma \quad \text{■ 2.10.2}$$

Corollary 2.10.3: If σ is logically true, then it is also Beth-provable:

$$\models \sigma \Rightarrow \vdash_B \sigma \quad \text{■ 2.10.3}$$

Definition 2.10.4: Let κ be a branch of a tableau, and let \mathcal{A} be an interpretation of \mathcal{L} . \mathcal{A} is said to **agree with** κ if:

- (i) $t\sigma$ is a node of $\kappa \Rightarrow \mathcal{A} \models \sigma$
- (ii) $f\sigma$ is a node of $\kappa \Rightarrow \mathcal{A} \not\models \sigma$ ■ 2.10.4

Lemma 2.10.5: Let T be a complete systematic tableau with $f\sigma$ at the origin, \mathcal{L} a language and \mathcal{A} the restriction of an interpretation of \mathcal{L} to the constant symbols occurring in σ , such that $\mathcal{A} \models \neg\sigma$. Then there is at least one branch of T which agrees with some extension of \mathcal{A} . ■ 2.10.5

Theorem 2.10.6: Soundness:

If sentence σ is Beth-provable by the set of sentences S of PrL, then σ is a consequence of S :

$$S \vdash_B \sigma \Rightarrow S \models \sigma \quad \text{■ 2.10.6}$$

Corollary 2.10.7: If sentence σ is Beth-provable, then it is also logically true:

$$\vdash_B \sigma \Rightarrow \models \sigma \quad \text{■ 2.10.7}$$

Theorem 2.10.8: Compactness:

A set of sentences S is satisfiable if and only if every subset of S is satisfiable.

■ 2.10.8

*Soundness and Completeness of Resolution Proofs***Theorem 2.10.9:** Soundness:

Let S be a set of clauses and $R^(S)$ the set of resolvents of S . If the empty clause belongs to $R^*(S)$, then S is not satisfiable:*

$$\square \in R^*(S) \Rightarrow S \text{ non-satisfiable} \quad \blacksquare \text{ 2.10.9}$$

Lemma 2.10.10: *If C'_1 and C'_2 are ground instances of clauses C_1 and C_2 , and if C' is the resolvent of C'_1 and C'_2 , then there exists a resolvent C of C_1 and C_2 , such that C' is a ground instance of C .*

■ 2.10.10

Theorem 2.10.11: Completeness:

Let S be a set of clauses and $R^(S)$ the set of resolvents of S . If S is not satisfiable, then the empty clause belongs to $R^*(S)$:*

$$S \text{ non-satisfiable} \Rightarrow \square \in R^*(S) \quad \blacksquare \text{ 2.10.11}$$

The intuitive interpretation of the above theorem is:

In order to prove satisfiability or non-satisfiability of a sentence by means of resolution, we just need to prove the empty clause by the corresponding set of clauses. Thus, if a consistent set of clauses S is given and if we wish to prove a sentence φ , we resolve the set of clauses $S \cup S'$ where S' corresponds to $\neg\varphi$. If we end up with the empty clause, the non-consistency is due to the assumption $\neg\varphi$, therefore φ is logically true.

Example 2.10.12: Assume the set of Horn clauses of Example 2.4.9. Give analytical answers using the method of resolution to the queries:

(a) “What can John steal?”

(b) “Can John steal Mary?”

Answer:

We reformulate the given Horn clauses and the queries in a set-theoretical form. We thus take the following set of Horn clauses.

$$C_1 : \{ \text{thief}(\text{Peter}) \}$$

$$C_2 : \{ \text{likes}(\text{Mary}, \text{food}) \}$$

$$C_3 : \{ \text{likes}(\text{Mary}, \text{wine}) \}$$

$$C_4 : \{ \text{likes}(\text{Peter}, \text{money}) \}$$

$$C_5 : \{ \text{likes}(\text{Peter}, x), \neg \text{likes}(x, \text{wine}) \}$$

$$C_6 : \{ \text{can_steal}(x, y), \neg \text{thief}(x), \neg \text{likes}(x, y) \}$$

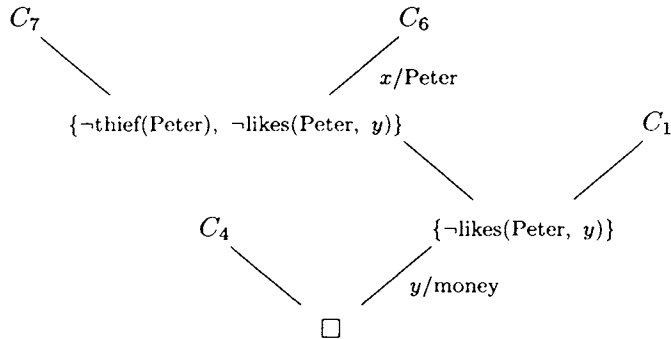
The queries take the form of claims:

$$C_7 : \{ \neg \text{can_steal}(\text{Peter}, y) \}$$

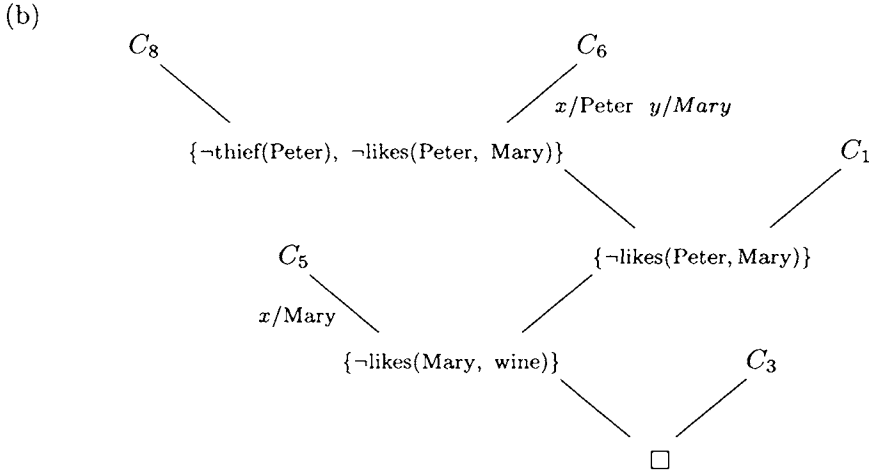
$$C_8 : \{ \neg \text{can_steal}(\text{Peter}, \text{Mary}) \}$$

Then:

(a)



In other words, starting from the negation of the query, we ended up with the empty clause, hence C_7 is not a true claim. What's more, the above proof gives us the value of the variable y , so answering query (a), $y = \text{money}$; which means Peter can steal money.



C_8 thus leads to the proof of the empty clause, hence Peter can steal Mary!

■ 2.10.12

Remark 2.10.13: If during resolution the unification algorithm finishes without yielding a Most General Unifier, in other words if we cannot prove \square from our data, then our goal is said to **fail**. In the opposite case, the goal is said to **succeed**, Remark 1.9.9. For instance, in example 2.10.12, the goal “can_steal(Peter, Mary)” succeeds.

■ 2.10.13

Completeness of the Axiomatic Proofs

The proof of the following completeness theorem of the axiomatic proofs method is beyond the context of this book. It can be found, e.g., in [Klee52, Mend64, Rasi74, RaSi70].

Theorem 2.10.14: Soundness and Completeness, Gödel, 1930:

A formula φ of PrL is derivable by the set of PrL sentences S , if and only if φ is a consequence of S . Formally:

$$S \vdash \varphi \Leftrightarrow S \models \varphi \quad \blacksquare \quad 2.10.14$$

Corollary 2.10.15: A formula φ of PrL is derivable by the axioms of PrL if and only if φ is logically true. Formally:

$$\vdash \varphi \Leftrightarrow \models \varphi \quad \blacksquare \quad 2.10.15$$

2.11 Decision Methods in Logic

Many mathematical problems come under a general problem scheme and can therefore be solved by means of a general procedure which is applied to a specific problem. For instance, we can answer the query

“does the polynomial $f(x)$ divide the polynomial $g(x)$?”

using the division algorithm; by dividing the specific $f(x)$ by the specific $g(x)$. If the remainder from the division is the zero polynomial, then the answer is “yes”. If the remainder differs from the zero polynomial, then the answer is “no”.

Definition 2.11.1: A method which allows us to answer “yes” or “no” to a specific case of a general query, is called a **decision procedure**. The problem of finding such a method for a general query is called a **decision problem** for the query. ■ 2.11.1

Many decision problems in mathematics cannot be solved in their general form, or have only specific solutions, which means that they are solved under certain conditions. The decision problem for a logic L is such a problem. A logic L is determined by its language, which consists of its logical and special symbols, and the axioms and rules by means of which we can prove and analyse well-formed sentences of L .

Definition 2.11.2: The decision problem for a logic L consists of finding an algorithmic method by means of which we can decide whether a well-formed sentence of L is derivable in L or not, in other words whether it is provable in L or not. ■ 2.11.2

The decision problem for PL is solved by the use of truth tables: if we are given a PL proposition A , we construct the truth table of A and check whether A is a tautology. If A is a tautology then by Corollary 1.12.2, A is derivable in PL; while if A is not a tautology, it is not derivable in PL. Hence the following theorem holds:

Theorem 2.11.3: *The decision problem for PL is solved.* ■ 2.11.3

The situation is not that simple for PrL. By Corollary 2.10.15 we know that a formula φ of a PrL language \mathcal{L} is derivable if and only if φ is logically true, i.e., if

it is true in all interpretations of \mathcal{L} . However the interpretations of a language \mathcal{L} are countless and we are naturally not in position to check them all. The following theorem has been known since 1936 [Chur36, Turi37].

Theorem 2.11.4: *The decision problem for PrL cannot be solved. In other words, there is no algorithmic method which allows us to decide whether a given PrL formula is derivable in PrL or not.* ■ 2.11.4

Although the decision problem for PrL cannot be solved in general, there are specific solutions [Klee52, Chur56]:

Theorem 2.11.5: *The decision problem for PrL can be solved for formulae in a Prenex Normal Form in which there is no existential quantifier preceding a universal quantifier. There is hence an algorithmic procedure which allows us to decide whether a formula of the form:*

$$\underbrace{(\forall x_1) \dots (\forall x_k)}_{\text{only } \forall} \underbrace{(\exists y_1) \dots (\exists y_\lambda)}_{\text{only } \exists} \varphi$$

is derivable in PrL or not.

■ 2.11.5

Theorem 2.11.6: *The decision problem for PrL can be solved for all formulae consisting exclusively of predicates of degree less or equal to 1, that is predicates with no more than one variable.* ■ 2.11.6

In [Chur56] there is an analytical table presenting all known specific solutions of the PrL decision problem.

While efforts to find specific solutions to the PrL decision problem were under way, there were attempts towards solving the decision problem for the satisfiability of a PrL formula, and thus determine an algorithmic method allowing us to determine whether or not a PrL formula is satisfiable. Theorems 2.6.6 (Loewenheim Skolem), 2.7.7 and 2.10.6 precisely state that the decision problem for the satisfiability of a PrL sentence admits specific solutions.

2.12 Exercises

2.12.1

Determine whether the following expressions are terms, formulae or none of these:

- | | |
|-------------------------------|---|
| (a) Nick | (f) $(\forall x) [\text{number}(x) \wedge x = x + x]$ |
| (b) $\text{Mathematician}(x)$ | (g) $= [(x + y), z]$ |
| (c) $\text{number}(6)$ | (h) $(x + y) + j^2$ |
| (d) $\text{is_a_planet}(x)$ | (i) the best book |
| (e) $(3 + 1) + 10$ | (j) $\text{hates}(x, y) \wedge \text{loves}(x, z)$ |

Solution:

- | | |
|-------------|-------------------|
| (a) term | (f) formula |
| (b) formula | (g) formula |
| (c) formula | (h) term |
| (d) formula | (i) none of these |
| (e) term | (j) formula |

2.12.2

Find the free occurrences of variables in the following formulae:

- (a) $(\forall x) P(x, y) \rightarrow (\forall z) Q(z, x)$
- (b) $Q(z) \rightarrow \neg(\forall x) (\forall y) P(x, y, a)$
- (c) $(\forall x) P(x) \wedge (\forall y) Q(x, y)$

Solution:

- (a) y, x (x also has a bound occurrence). (b) z . (c) x .

2.12.3

Determine the free occurrences of variables in the following formulae. Which of these formulae are sentences?

- (a) $(\forall x)(\forall y)(\forall z)[x > y \wedge y > z] \rightarrow (\exists w)[w > w]$
- (b) $(\exists x)(\text{is_red}(x)) \vee (\forall y)[\text{is_blue}(y) \vee \text{is_yellow}(x)]$
- (c) $x + x = x + x$
- (d) $(\exists y)[x + x = x + x]$
- (e) $(\exists x)(\exists y)[\text{is_teacher}(x, y) \wedge \text{teaches}(x, y, z)]$

Solution:

- (a) No free occurrence. This formula is a sentence.
- (b) x has a free occurrence in the second disjunctive subformula.
- (c) x occurs free.
- (d) x occurs free.
- (e) z occurs free.

2.12.4

Using the arithmetical symbol “ $<$ ” (less than) and the PrL language, formulate the following sentences:

- (a) There exists a number x less than 5 and greater than 3.
- (b) For every number x , there exists a number y smaller than x .
- (c) For every number x , there exists a number y greater than x .
- (d) For all numbers x and y , sums $x + y$ and $y + x$ are equal.
- (e) For every number x , there exists a number y such that for every z , for which, if the subtraction $z - 5$ is less than y , then the subtraction $x - 7$ is less than 3.

Solution:

We introduce functions $-(x, y)$, $+(x, y)$ in order to express the subtraction and the sum of x, y , and we also introduce the predicate “number(x)” to express the

fact that x is a number. Then the sentences are written as follows:

- (a) $(\exists x) [\text{number}(x) \wedge < (x, 5) \wedge < (3, x)]$
- (b) $(\forall x) (\exists y) [\text{number}(x) \wedge \text{number}(y) \wedge < (y, x)]$
- (c) $(\forall x) (\exists y) [\text{number}(x) \wedge \text{number}(y) \wedge < (x, y)]$
- (d) $(\forall x) (\forall y) [\text{number}(x) \wedge \text{number}(y) \rightarrow (+ (x, y), + (y, x))]$
- (e) $(\forall x) (\exists y) (\forall y)$
 $[\text{number}(x) \wedge \text{number}(y) \wedge (y) \wedge (< (- (z, 5), y) \rightarrow < (- (x, 7), 3))]$

2.12.5

If θ, ψ, ξ are substitutions, E the identical substitution, and σ an atomic formula, prove that:

- (a) $\theta E = E \theta = \theta$
- (b) $(\sigma \theta) \psi = \sigma (\theta \psi)$
- (c) $(\theta \psi) \xi = \theta (\psi \xi)$

Solution:

- (b) Consider $\theta = \{u_1/s_1, \dots, u_m/s_m\}$, $\psi = \{v_1/t_1, \dots, v_n/t_n\}$.

We first prove that for all terms τ , the following holds:

$$(\tau \theta) \psi = \tau (\theta \psi) \quad (*)$$

The proof uses induction on the length of τ :

If τ is a constant, then $(*)$ obviously holds.

If τ is a variable and

if $x \notin \{u_1, \dots, u_m\} \cup \{v_1, \dots, v_n\}$, then $x(\theta \psi) = x = (x \theta) \psi$.

if $x \in \{u_1, \dots, u_m\}$, then there exists some $1 \leq i \leq m$ such that $(x \theta) \psi = s_i \psi = x(\theta \psi)$.

if $x \in \{v_1, \dots, v_n\} - \{u_1, \dots, u_m\}$, then there exists some $1 \leq j \leq n$ such that $(x \theta) \psi = v_j = x(\theta \psi)$.

If t is the term $f(u_1, \dots, u_k)$, then $(*)$ holds by the induction assumption and by Definition 2.2.19. If now σ is an atomic formula then, by $(*)$ and Definition 2.2.19 (why?), $(\sigma \theta) \psi = \sigma (\theta \psi)$ holds.

2.12.6

Determine whether the following pairs of formulae are variations:

$$(a) \quad P(f(x, y), g(z), a), \quad P(f(y, x), g(u), a)$$

$$(b) \quad P(x, x), \quad P(x, y)$$

Solution:

$$(a) \quad \text{yes (why?)} \qquad (b) \quad \text{no (why?)}$$

2.12.7

Prove that:

$$(a) \quad \text{If } \text{free}(x, a, A), \text{ then } \vdash A(x/a) \rightarrow (\exists x)A$$

$$(b) \quad \vdash (\forall x) A \rightarrow A$$

$$(c) \quad \vdash A \rightarrow (\exists x) A$$

$$(d) \quad \vdash (\forall x) A \rightarrow (\exists x) A$$

$$(e) \quad \vdash (\forall x) (\exists y) (x = y)$$

$$(f) \quad \vdash (\forall x) (\forall y) (x = y \rightarrow y = x)$$

$$(g) \quad \vdash (\forall x) (\forall y) (\forall z) [(x = y \wedge y = z) \rightarrow x = z]$$

Solution:

$$(a) \quad \text{If } \text{free}(x, a, A), \text{ then also } \text{free}(x, a, \neg A). \text{ By axiom (4) we have:}$$

$$\vdash (\forall x) \neg A \rightarrow \neg A(x/a)$$

Then by axiom (3) and Modus Ponens we conclude:

$$\vdash \neg \neg A(x/a) \rightarrow \neg (\forall x) \neg A$$

By tautology $\neg \neg A \leftrightarrow A$, Remark 2.3.4 (4), and the theorem of substitution of equivalences we have:

$$\vdash A(x/a) \rightarrow \neg (\forall x) \neg A$$

Then also:

$$\vdash A(x/a) \rightarrow (\exists x) A$$

(b) By axiom (4), since $\text{free}(x, a, A)$.

(d) $\vdash A \rightarrow (\exists x) A$ by (c), and $\vdash (\forall x) \rightarrow A$ by (b). Then by tautology:

$$(B \rightarrow C) \rightarrow [(A \rightarrow B) \rightarrow (A \rightarrow C)]$$

we have:

$$\vdash (A \rightarrow (\exists x) A) \rightarrow [((\forall x) A \rightarrow A) \rightarrow ((\forall x) A \rightarrow (\exists x) A)]$$

A double Modus Ponens application leads to what we are seeking.

(e) Assume formula $x = y$ is A . Then by (a) we have:

$$\vdash x = x \rightarrow (\exists y) (x = y)$$

By axiom (6), Remark 2.3.7, and Modus Ponens we have $\vdash (\exists y) (x = y)$.

Then by the rule of generalization, $\vdash (\forall x) (\exists y) (x = y)$.

(f) By axiom (7) we have:

$$\vdash x = y \rightarrow (x = x \rightarrow y = x)$$

We apply tautology:

$$[A \rightarrow (B \rightarrow C)] \leftrightarrow [B \rightarrow (A \rightarrow C)]$$

which gives:

$$\vdash x = x \rightarrow (x = y \rightarrow y = x)$$

By axiom (6) and Modus Ponens, we have $\vdash x = y \rightarrow y = x$. A double Modus Ponens application leads to what we are seeking.

(g) By axiom (7), we have:

$$\vdash y = x \rightarrow (y = z \rightarrow x = z)$$

and by (f):

$$\vdash x = y \rightarrow (y = z \rightarrow x = z)$$

We apply tautology:

$$[A \rightarrow (B \rightarrow C)] \leftrightarrow [(A \wedge B) \rightarrow C]$$

which gives:

$$\vdash (x = y \wedge y = z) \rightarrow x = z.$$

We then apply Modus Ponens three times.

2.12.8

Determine the set-theoretical form of the following sentences:

- (a) John loves food.
- (b) Apples are food.
- (c) Chicken is food.
- (d) Whatever can be eaten without killing somebody is food.
- (e) Bill eats and is still alive.
- (f) Mary eats whatever Bill eats.

Solution:

We introduce the following predicates:

Loves(x, y) : x loves y	Food(x) : x is one kind of food
Eats(x, y) : x eats y	Lives(x) : x is alive

Then

$$(a) \quad (\forall x) [\text{Food}(x) \rightarrow \text{Loves}(\text{John}, x)] \leftrightarrow (\forall x) [\neg \text{Food}(x) \vee \text{Loves}(\text{John}, x)]$$

$$\text{Hence } S_a = \{\{\neg \text{Food}(x), \text{Loves}(\text{John}, x)\}\}.$$

$$(b) \quad \text{Food}(\text{Apples}). \quad \text{Hence } S_b = \{\{\text{Food}(\text{Apples})\}\}.$$

$$(d) \quad (\forall x) (\forall y) [\text{Eats}(x, y) \wedge \text{Lives}(x) \rightarrow \text{Food}(y)]$$

$$\leftrightarrow (\forall x) (\forall y) [\neg \text{Eats}(x, y) \vee \neg \text{Lives}(x) \vee \text{Food}(y)]$$

$$\text{Hence } S_d = \{\{\neg \text{Eats}(x, y), \neg \text{Lives}(x), \text{Food}(y)\}\}.$$

2.12.9

Assume dragons really exist and we capture a big one. Formulate the following sentences of everyday speech using Horn clauses.

- (a) Every dragon leaving in the zoo is not happy.
- (b) Every animal that meets polite people is happy.
- (c) People visiting the zoo are polite.
- (d) The animals living in the zoo meet the people that visit the zoo.

Solution:

We introduce the following predicates:

$\text{Dragon}(x)$: x is a dragon	$\text{Person}(x)$: x is a person
$\text{Happy}(x)$: x is happy	$\text{Lives}(x, y)$: x lives in y
$\text{Animal}(x)$: x is an animal	$\text{Nice}(x)$: x is polite
$\text{Visits}(x, y)$: x visits y	$\text{Meets}(x, y)$: x meets y

Then:

- (a) $\leftarrow \text{Dragon}(x), \text{Happy}(x), \text{Lives}(x, \text{Zoo}).$
- (b) $\text{Happy}(x) \leftarrow \text{Zoo}(x), \text{Person}(y), \text{Nice}(y), \text{Meets}(x, y).$
- (c) $\text{Nice}(x) \leftarrow \text{Person}(x), \text{Visits}(x, \text{Zoo}).$
- (d) $\text{Meets}(x, y) \leftarrow \text{Zoo}(x), \text{Person}(y), \text{Lives}(x, \text{Zoo}), \text{Visits}(y, \text{Zoo}).$

2.12.10

Formulate the following sentences of everyday speech with Horn clauses:

- (a) x is the mother of y , if x is a woman and a parent of y .
- (b) x is the father of y , if x is a man and a parent of y .
- (c) x is human, if his parent is a human.
- (d) x is a human, if his father is a human.
- (e) Nobody is his own parent.

2.12.11

We have two empty receptacles of 5 and 7 litres respectively. We wish to determine a sequence of actions which would result in leaving 4 litres of water in the 7 litres receptacle. Only two actions are allowed:

- (1) Fill up a receptacle.
- (2) Transfer water from one receptacle to the other until the corresponding receptacle is filled or empty.

Formulate the problem and the allowed actions with Horn clauses.

Solution:

We introduce the predicate

Contain(u, v) : The 7 litre receptacle contains u litres and
the 5 litre receptacle contains v litres.

If we assume that relations $x + y = z$ and $x \leq y$ have already been defined, then the problem can be expressed with the following clauses:

- C_1 : Contain(0, 0) \leftarrow
- C_2 : \leftarrow Contain(4, y)
- C_3 : Contain(7, y) \leftarrow Contain(x, y)
- C_4 : Contain($x, 5$) \leftarrow Contain(x, y)
- C_5 : Contain(0, y) \leftarrow Contain(x, y)
- C_6 : Contain($x, 0$) \leftarrow Contain(x, y)
- C_7 : Contain(0, y) \leftarrow Contain(u, v), $u + v = y$, $y \leq 5$
- C_8 : Contain($x, 0$) \leftarrow Contain(u, v), $u + v = x$, $x \leq 7$
- C_9 : Contain(7, y) \leftarrow Contain(u, v), $u + v = w$, $7 + y = w$
- C_{10} : Contain($x, 5$) \leftarrow Contain(u, v), $u + v = w$, $5 + x = w$

C_1 expresses the initial situation and C_2 the goal. C_3 and C_4 express the filling of the first and the second receptacles respectively.

C_5 and C_6 express the emptying of the first and the second receptacles, C_7 and C_8 the transfer of water until the first receptacle is empty and C_9 and C_{10} until the second one is empty.

2.12.12

Let $\mathcal{L} = \{a, b, P\}$ be a PrL language and let \mathcal{A} be an interpretation of \mathcal{L} with a universe $D = \{a, b\}$, such that (a, a) and (b, b) belong to $\varepsilon(P)$, whereas (a, b) and (b, a) do not belong to $\varepsilon(P)$. Find out whether the following formulae are true in \mathcal{A} .

- | | |
|--|--------------------------------------|
| (a) $(\forall x)(\exists y) P(x, y)$ | (d) $(\forall x)(\forall y) P(x, y)$ |
| (b) $(\exists x)(\forall y) P(x, y)$ | (e) $(\exists y) \neg P(a, y)$ |
| (c) $(\forall x)(\exists y) [P(x, y) \rightarrow P(y, x)]$ | (f) $(\forall x) P(x, x)$ |

Solution:

We declare fact $(a, b) \in \varepsilon(P) \subseteq D^2$ allocating value t , true, to $P(a, b)$, and we declare $(a, b) \notin \varepsilon(P) \subseteq D^2$ allocating value f , false, to $P(a, b)$. Then, for the given interpretation, we can use the following table of values of P in relation to the universe of the interpretation:

$P(a, a)$	$P(a, b)$	$P(b, a)$	$P(b, b)$
t	f	f	t

- (a) If x takes value a or b , then by the above table we can see that there are values of y (a or b respectively) such that the corresponding $P(x, y)$ is satisfied. Hence $(\forall x)(\exists y)P(x, y)$ is true in \mathcal{A} .
- (b) False (why?).
- (c) We create a table giving us the values of (c) for all the possible values of x, y :

(x, y)	(y, x)	$P(x, y)$	\longrightarrow	$P(y, x)$
(a, a)	(a, b)	t	t	t
(a, b)	(b, a)	f	t	f
(b, a)	(a, b)	f	t	f
(b, b)	(b, b)	t	t	f

Then (c) is true in \mathcal{A} .

- (d) When x is interpreted as a and y as b , $P(x, y)$ is obviously not true in \mathcal{A} .
- (e) $P(a, y)$ is true when y takes value b .

2.12.13

Let $\sigma : (\exists x)P(x) \rightarrow (\forall x)P(x)$ be a sentence.

- (a) Prove that σ is always true in interpretation with a universe which is a singleton.
- (b) Find an interpretation with a universe consisting of two elements in which σ is not true.

Solution:

- (a) Let $\mathcal{A} = (\{a\}, \varepsilon(P))$ be any interpretation of the language $\mathcal{L} = \{P\}$ in which σ is expressed. Then by Definition 2.5.5 we have:

$$\begin{aligned}\mathcal{A} \models (\exists x) P(x) \rightarrow (\forall x) P(x) &\Leftrightarrow \mathcal{A} \models (\forall x) P(x) \quad \text{or} \quad \mathcal{A} \not\models (\exists x) P(x) \\ &\Leftrightarrow \mathcal{A} \models P(a) \quad \text{or} \quad \mathcal{A} \models \neg P(a) \\ &\Leftrightarrow \mathcal{A} \models P(a) \vee \neg P(a)\end{aligned}$$

But the last disjunction is true because of tautology $A \vee \neg A$, Remark 2.3.4.

- (b) Let $\{a, b\}$ be the universe of the interpretation we are seeking, with $a \neq b$. For σ to be false in \mathcal{A} , we need $\mathcal{A} \not\models (\forall x) P(x)$ and $\mathcal{A} \models (\exists x) P(x)$. We thus construct the following table of values of P for a and b :

$P(a)$	$P(b)$
t	f

In this interpretation σ is not true (why?).

2.12.14

Find an interpretation with a universe consisting of two elements $\{a, b\}$ such that sentence $(\exists x)(\exists y) P(x, y)$ is true and sentence $(\forall x)(\exists y) P(x, y)$ is false.

2.12.15

Let $\mathcal{L} = \{\Delta, c_1, c_2, \dots, c_9\}$ be a language, where Δ is a 2-ary predicate symbol and c_1, \dots, c_9 are constant symbols, and $\mathcal{A} = (\{1, 2, \dots, 9\}, /)$ be an interpretation, where $/$ is the divisibility relation, $\varepsilon(\Delta) = /$ and $\varepsilon(c_i) = i$, $i = 1, \dots, 9$. Determine which of these sentences are true in this interpretation. The answer has to be justified.

- | | |
|--|---|
| (a) $(\forall y) \Delta(c_1, y)$ | (c) $(\exists x)(\forall y) \Delta(x, y)$ |
| (b) $(\forall x)[\Delta(x, c_5) \leftrightarrow (x = c_1) \vee (x = c_5)]$ | (d) $(\exists x)(\forall y) \Delta(y, x)$ |

Solution:

By Definition 2.5.5, we have:

- | | |
|-----------------|----------------------------|
| (a) true (why?) | (c) true (for which x ?) |
| (b) true (why?) | (d) false (why?) |

2.12.16

Let $\mathcal{L} = \{\leq, =, +, \cdot, 0, 1\}$ be the language of arithmetic.

- (a) Write in \mathcal{L} a sentence $P(x)$ interpreted in the usual mathematical interpretation as “ x is a first number”.
- (b) Using $P(x)$ as a predicate, write a sentence of \mathcal{L} expressing the fact that the sum of two prime numbers others than 2 is an even number.

Solution:

- (a) x is a prime number if and only if x and 1 are the only divisors of x . Then:

$$P(x) \leftrightarrow (\forall y)(\forall z)[(x = y \cdot z) \rightarrow (x = y \wedge z = 1) \vee (x = z \wedge y = 1)]$$

- (b) $(\forall x)(\forall y)[(P(x) \wedge P(y)) \rightarrow (\exists z)(+(x \cdot x, y \cdot y) = +(z, z))]$

2.12.17

A PrL language $\mathcal{L} = \{=, \leq\}$ is given, as well as a set of axioms such that:

$$\begin{aligned} \Sigma = \{ & (\forall x)[x \leq x], & (\forall x)(\forall y)[(x \leq y) \wedge (y \leq x) \rightarrow (x = y)], \\ & (\forall x)(\forall y)(\forall z)[(x \leq y) \wedge (y \leq z) \rightarrow (x \leq z)], \\ & (\forall x)(\forall y)[\neg(x = y) \wedge (x \leq y) \rightarrow \\ & \quad (\exists z)[(x \leq z) \wedge (z \leq y) \wedge \neg(x = z) \wedge \neg(y = z)]], \\ & (\forall x)(\forall y)[(x = y) \vee (x \leq y) \vee (y \leq x)] \} \end{aligned}$$

Determine two different interpretations of \mathcal{L} , \mathcal{A}_1 and \mathcal{A}_2 , such that $\mathcal{A}_1 \models \Sigma$ and $\mathcal{A}_2 \not\models \Sigma$.

Solution:

The first 3 axioms of Σ axiomatize the order, the fourth declares that the order is dense and the fifth axiom states that the order is total. We know that real, rational and natural numbers with the usual ordering are totally ordered and, furthermore, the order is dense for real and rational numbers whereas it is not dense for natural numbers (why?).

2.12.18

Let $\mathcal{A}_1 = (\mathbb{Q}, \varepsilon_1(P))$ and $\mathcal{A}_2 = (\mathbb{N}, \varepsilon_2(P))$ be two interpretations of the PrL language $\mathcal{L} = \{P\}$, where $\varepsilon_1(P)$ and $\varepsilon_2(P)$ are the usual orders of rational numbers \mathbb{Q} and natural numbers \mathbb{N} , with $\theta(\mathcal{A}_1)$ and $\theta(\mathcal{A}_2)$ the corresponding theories (Definition 2.5.10).

- (a) Determine two sentences σ_1 and σ_2 of \mathcal{L} such that

$$\mathcal{A}_1 \models \sigma_1 \wedge \sigma_2 \quad \text{and} \quad \mathcal{A}_2 \not\models \sigma_1 \vee \sigma_2$$

- (b) Determine which of the following are valid and justify the answer:

$$\theta(\mathcal{A}_1) = \theta(\mathcal{A}_2), \quad \theta(\mathcal{A}_1) \not\subseteq \theta(\mathcal{A}_2), \quad \theta(\mathcal{A}_2) \subseteq \theta(\mathcal{A}_1)$$

Solution:

- (a) By Definition 2.5.5, both σ_1 and σ_2 have to be valid in \mathcal{A}_1 , whereas in \mathcal{A}_2 , σ_1 and σ_2 cannot be valid. We take for σ_1 the sentence expressing dense orders:

$$\begin{aligned} \sigma_1 : (\forall x) (\forall y) \Big[\neg(x = y) \wedge (x \leq y) \\ \rightarrow (\exists z) [(x \leq z) \wedge (z \leq y) \wedge \neg(x = z) \wedge \neg(y = z)] \Big] \end{aligned}$$

Another characteristic difference between \mathbb{Q} and \mathbb{N} is the existence of a least element in \mathbb{N} , whereas \mathbb{Q} has no least element. Let σ_2 be the sentence stating that there is no least element:

$$\sigma_2 : \neg(\exists x) (\forall y) P(x, y)$$

Then $\mathcal{A} \models \sigma_1 \wedge \sigma_2$ and $\mathcal{A}_2 \not\models \sigma_1 \vee \sigma_2$ (why?)

(b) $\mathcal{A}_1 \models \sigma_1$ but $\mathcal{A}_2 \not\models \sigma_1$. Hence $\theta(\mathcal{A}_1) \neq \theta(\mathcal{A}_2)$.

$\mathcal{A}_2 \models \neg\sigma_2$ (why?), but $\mathcal{A}_1 \not\models \neg\sigma_2$. Hence $\theta(\mathcal{A}_1) \not\subseteq \theta(\mathcal{A}_2)$.

$\mathcal{A}_2 \models \neg\sigma_1$ (why?), but $\neg\sigma_1 \notin \theta(\mathcal{A}_1)$ (why?). Hence $\theta(\mathcal{A}_2) \not\subseteq \theta(\mathcal{A}_1)$.

2.12.19

Convert into Prenex Normal Forms the following sentences:

(a) $(\forall x) P(x) \rightarrow (\exists x) Q(x)$

(b) $(\forall x) (\forall y) [(\exists z) P(x, y) \wedge P(x, z)] \rightarrow (\exists u) Q(x, y, u)$

(c) $(\forall x) (\forall y) [(\exists z) P(x, y, z) \wedge [(\exists u) Q(x, u) \rightarrow (\exists u) Q(y, u)]]$

Solution:

We use formulae (1) to (6), (9) and (10) of section 2.5.

$$\begin{aligned}
 (c) \quad & (\forall x) (\forall y) [(\exists z) P(x, y, z) \wedge [(\exists u) Q(x, u) \rightarrow (\exists u) Q(y, u)]] \\
 \leftrightarrow & (\forall x) (\forall y) [(\exists z) P(x, y, z) \wedge [\neg(\exists u) Q(x, u) \vee (\exists u) Q(y, u)]] \\
 \leftrightarrow & (\forall x) (\forall y) [(\exists z) P(x, y, z) \wedge [(\forall u) \neg Q(x, u) \vee (\exists w) Q(y, w)]] \\
 \leftrightarrow & (\forall x) (\forall y) (\exists z) [P(x, y, z) \wedge (\forall u) (\exists w) [\neg Q(x, u) \vee Q(y, w)]] \\
 \leftrightarrow & (\forall x) (\forall y) (\exists z) (\forall u) (\exists w) [P(x, y, z) \wedge (\neg Q(x, u) \vee Q(y, w))]
 \end{aligned}$$

2.12.20

Determine the set-theoretical form of the following sentences:

$$\begin{aligned}
 \sigma : \quad & (\exists x) (\forall y) (\exists z) [(P(x, y) \vee \neg Q(x) \vee R(z)) \\
 & \quad \wedge (\neg P(x, y) \vee \neg Q(x)) \wedge (\neg P(x, y) \vee R(z))]
 \end{aligned}$$

$$\sigma' : \neg(\forall x) (\exists y) [P(x, y) \rightarrow Q(y)]$$

$$\varphi : \neg[(\forall x) P(x) \rightarrow (\exists y) (\forall z) Q(y, z)]$$

Solution:

σ is already in a Prenex Normal Form. Hence we need to determine the Skolem Normal Form SNF.

$$\begin{aligned}\sigma^* : \quad & (\forall y) (\exists z) [(P(a, y) \vee \neg Q(a) \vee R(z)) \\ & \quad \wedge (\neg P(a, y) \vee \neg Q(a)) \wedge (\neg P(a, y) \vee R(z))] \quad (\text{why?}) \\ \leftrightarrow & (\forall y) [(P(a, y) \vee \neg Q(a) \vee R(f(y))) \\ & \quad \wedge (\neg P(a, y) \vee \neg Q(a)) \wedge (\neg P(a, y) \vee R(f(y)))]\end{aligned}$$

where a is a new constant and $f(y)$ is a new function symbol of one variable. Then by Definition 2.6.10 the set-theoretical form of σ is:

$$S_\sigma = \{\{P(a, y), \neg Q(a), R(f(y))\}, \{\neg P(a, y), \neg Q(a)\}, \{\neg P(a, y), R(f(y))\}\}$$

φ must now be converted into a PNF.

$$\begin{aligned}\varphi & \leftrightarrow \neg [\neg(\forall x) P(x) \vee (\exists y) (\forall z) Q(y, z)] \\ & \leftrightarrow (\forall x) P(x) \wedge \neg(\exists y) (\forall z) Q(y, z) \\ & \leftrightarrow (\forall x) P(x) \wedge (\forall y) (\exists z) \neg Q(y, z) \\ & \leftrightarrow (\forall x) (\forall y) [P(x) \wedge (\exists z) \neg Q(y, z)] \\ & \leftrightarrow (\forall x) (\forall y) (\exists z) [P(x) \wedge \neg Q(y, z)]\end{aligned}$$

The corresponding Skolem Form is:

$$\varphi^* : (\forall x) (\forall y) [P(x) \wedge \neg Q(y, f(x, y))]$$

where $f(x, y)$ is a new function symbol of two variables. The set-theoretical form of σ is:

$$S_\sigma = \{\{P(x)\}, \{\neg Q(y, f(x, y))\}\}$$

2.12.21

The following sentences are given:

F_1 : Whoever saves up money earns the interest.

F_2 : If there is no interest, then nobody saves up money.

We introduce the following predicates:

$A(x, y)$: x saves up y $T(x)$: x is interest

$K(x, y)$: x earns y $X(x)$: x is money

Formulate F_1 and F_2 in the language defined by the above predicates and determine the Skolem Forms of F_1 and $\neg F_2$.

2.12.22

Let $(G, \#)$ be a group. Describe with carefully selected predicates the properties of G , in other words the fact that G is closed for $\#$, $\#$ is associative and has an identity element and inverse elements. Determine the SNF of the formulae which express these properties.

Solution:

We introduce predicate $P(x, y, z)$ which is interpreted as $x\#y = z$. Then, to express the fact that G is closed for $\#$ we use formula

$$\sigma_1 : (\forall x) (\forall y) (\exists z) P(x, y, z)$$

Associativity is expressed by formula:

$$\begin{aligned} \sigma_2 : & (\forall x) (\forall y) (\forall u) (\forall v) (\forall w) [P(x, y, u) \wedge P(y, z, v) \wedge P(u, z, w) \rightarrow P(x, v, w)] \\ & \wedge (\forall x) (\forall y) (\forall z) (\forall u) (\forall v) (\forall w) [P(x, y, u) \wedge P(y, z, v) \wedge P(x, v, w) \rightarrow P(u, z, w)] \end{aligned}$$

The existence of an identity element:

$$\sigma_3 : (\exists y) (\forall x) [P(x, y, x) \wedge P(y, x, x)]$$

The existence of an inverse element:

$$\sigma_4 : (\exists y) (\forall x) (\exists z) [P(x, y, x) \wedge P(y, x, x) \rightarrow P(x, z, y) \wedge P(z, x, y)]$$

The SNF of these sentences are:

$$\sigma_1^* : (\forall x) (\forall y) P(x, y, f(x, y))$$

where f is a new function of two variables,

$$\begin{aligned} \sigma_2^* : & (\forall x) (\forall y) (\forall z) (\forall u) (\forall v) (\forall w) \\ & \left[[P(x, y, u) \wedge P(y, z, v) \wedge P(u, z, w) \rightarrow P(x, v, w)] \right. \\ & \quad \left. \wedge [P(x, y, u) \wedge P(y, z, v) \wedge P(x, v, w) \rightarrow P(u, z, w)] \right] \quad (\text{why?}) \\ \leftrightarrow & (\forall x) (\forall y) (\forall z) (\forall u) (\forall v) (\forall w) \\ & \left[[P(x, y, u) \wedge P(y, z, v) \wedge P(u, z, w) \rightarrow P(x, v, w)] \right. \\ & \quad \left. \wedge [P(x, y, u) \wedge P(y, z, v) \wedge P(x, v, w) \rightarrow P(u, z, w)] \right] \\ \leftrightarrow & (\forall x) (\forall y) (\forall z) (\forall u) (\forall v) (\forall w) \\ & \left[[\neg P(x, y, u) \vee \neg P(y, z, v) \vee \neg P(u, z, w) \vee P(x, v, w)] \right. \\ & \quad \left. \wedge [\neg P(x, y, u) \vee \neg P(y, z, v) \vee \neg P(x, v, w) \vee P(u, z, w)] \right] \end{aligned}$$

which coincides with the PNF of σ_2 (why?),

$$\sigma_3^* : (\forall x) [P(x, a, x) \wedge P(a, x, x)]$$

$$\begin{aligned} \sigma_4^* : & (\forall x) (\exists z) [\neg P(x, b, x) \vee \neg P(b, x, x) \vee P(x, z, b) \vee P(z, x, b)] \\ \leftrightarrow & (\forall x) [\neg P(x, b, x) \vee \neg P(b, x, x) \vee P(x, g(x), b) \vee P(g(x), x, b)] \end{aligned}$$

where a and b are new constants and g is a new function.

Another possible way to deal with the problem would be to use a PrL language \mathcal{L} , where $\mathcal{L} = \{=, f, i, c\}$, f being a symbol of a function of two variables with $f(x, y) = x \sharp y$ as the intuitive interpretation, i a function of one variable, where $i(x)$ is the inverse of x , e a constant symbol, the identity element of the group, and $=$, the equality predicate symbol, for example [Hami78].

2.12.23

Let S_1 and S_2 be two sets of clauses:

$$\begin{aligned} S_1 &= \{\{P(a)\}, \{P(x), P(f(x))\}\} \\ S_2 &= \{\{P(x), Q(x)\}, \{R(z)\}, \{T(y), \neg W(y)\}\} \end{aligned}$$

Determine the Herbrand sets H_3 for S_1 , and H_1, \dots, H_{10} for S_2 .

Solution:

For S_2 : There are no variables occurring in S_2 . We thus introduce c , a constant symbol. Then $H_0 = \{c\}$. However, there are no function symbols occurring in S_2 .

$$\text{Then } H_0 = H_1 = \dots = H_{10} = \{c\}.$$

2.12.24

Determine the Herbrand sets H_0 and H_1 for the set of clauses

$$S = \{\{P(f(x), a, g(f(x), b))\}\}$$

Determine all the ground instances (Definition 2.4.3) of S in H_0 and H_1 .

Solution:

$$H_0 = \{a, b\}$$

$$H_1 = \{a, b, f(a), f(b), g(f(a), b), g(f(b), b)\}$$

Then the instances of S for H_0 are:

$$S_{11} = \{\{P(f(a), a, g(f(a), b))\}\}$$

$$S_{12} = \{\{P(f(b), a, g(f(b), b))\}\}$$

The instances of S for H_1 are:

$$S_{21} = \{\{P(f(a), a, g(f(a), b))\}\}$$

$$S_{22} = \{\{P(f(b), a, g(f(b), b))\}\}$$

$$S_{23} = \{\{P(f(g(f(a), b)), a, g(f(g(f(a), b)), b))\}\}$$

$$S_{24} = \{\{P(f(g(f(b), b)), a, g(f(g(f(b), b)), b))\}\}$$

2.12.25

Prove by means of Complete Systematic Tableaux the following sentences:

$$(a) \quad [(\exists x) P(x) \rightarrow (\forall x) Q(x)] \rightarrow (\forall x) [P(x) \rightarrow Q(x)]$$

$$(b) \quad \neg(\forall x) P(x) \leftrightarrow (\exists x) (\neg P(x))$$

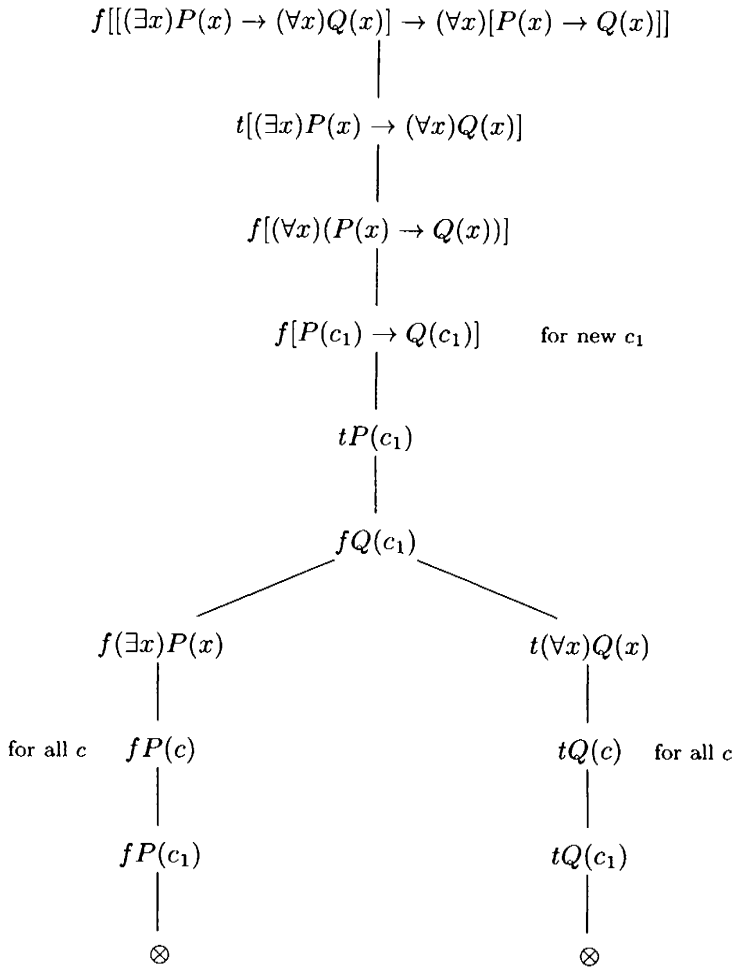
$$(c) \quad (\forall x) (P(x) \wedge Q(x)) \leftrightarrow ((\forall x) P(x) \wedge (\forall x) Q(x))$$

$$(d) \quad (\exists x) (P(x) \vee Q(x)) \leftrightarrow (\exists x) P(x) \vee (\exists x) Q(x)$$

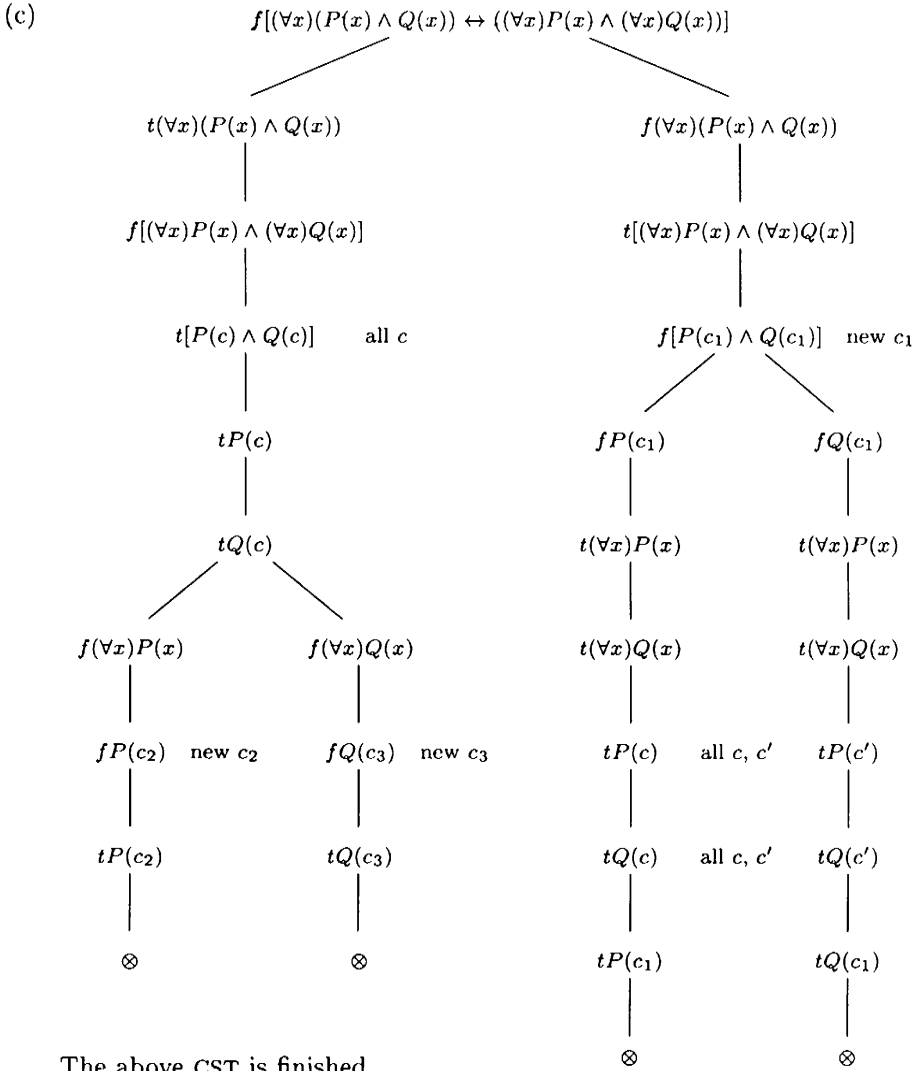
$$(e) \quad (\exists x) (\forall y) P(x, y) \rightarrow (\forall y) (\exists x) P(x, y)$$

Solution:

(a)



Then by Definition 2.3.7, (a) is Beth-provable, in other words it has a proof by means of CST.



The above CST is finished

and contradictory, hence it constitutes a proof of (c).

2.12.26

Prove by means of CST that the following sentences are not Beth-provable:

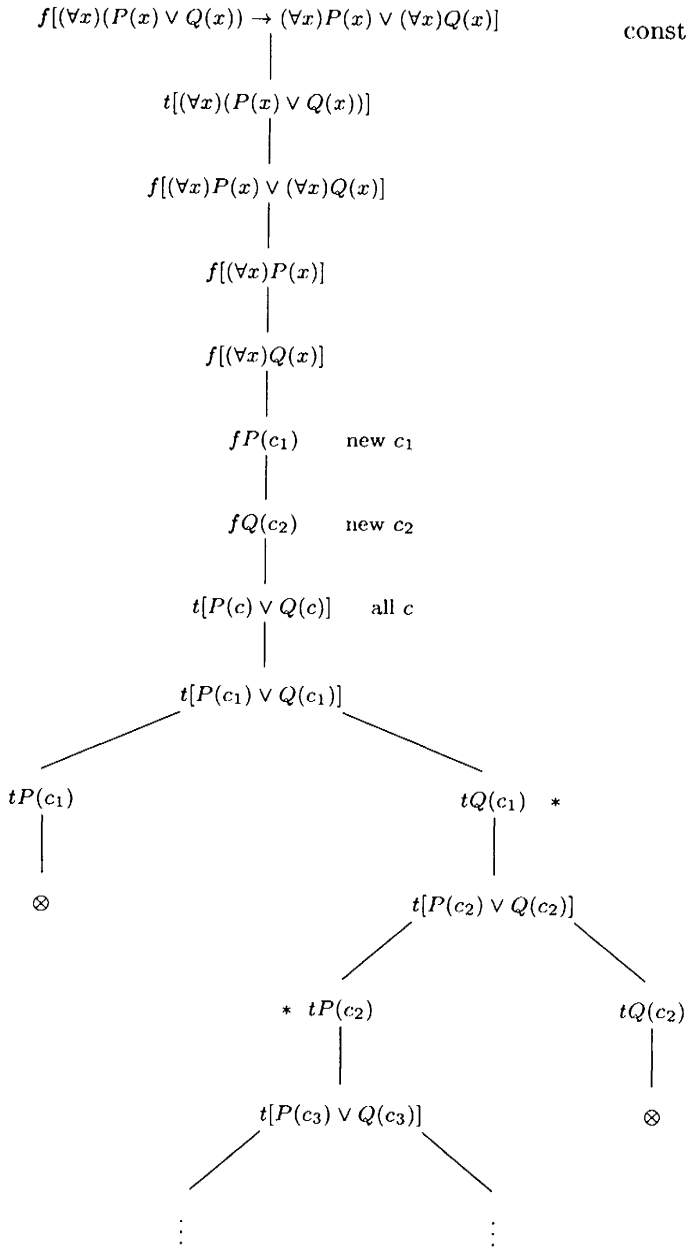
$$\sigma : (\forall x)(P(x) \vee Q(x)) \rightarrow (\forall x)P(x) \vee (\forall x)Q(x)$$

$$\varphi : (\exists x)P(x) \wedge (\exists x)Q(x) \rightarrow (\exists x)(P(x) \wedge Q(x))$$

Determine two interpretations \mathcal{A}_σ and \mathcal{A}_φ in which σ and φ respectively are not true.

Solution:

Assume σ is Beth-provable. Then the CST of σ with an $f\sigma$ origin must be contradictory. We construct the CST:



This CST is however non-contradictory (why?), so σ cannot be Beth-provable.

In order to determine an interpretation \mathcal{A}_σ , notice, in the non-contradictory branches, nodes tP and tQ marked with $*$ (why?). Constants c_1 and c_2 constitute the universe of the interpretation. We impose $\varepsilon(P) = \{c_1\}$ and $\varepsilon(Q) = \{c_2\}$. Then σ is not true in $\mathcal{A}_\sigma = (\{c_1, c_2\}, \varepsilon(P), \varepsilon(Q))$ (give a complete proof of this last claim according to Definition 2.5.5).

2.12.27

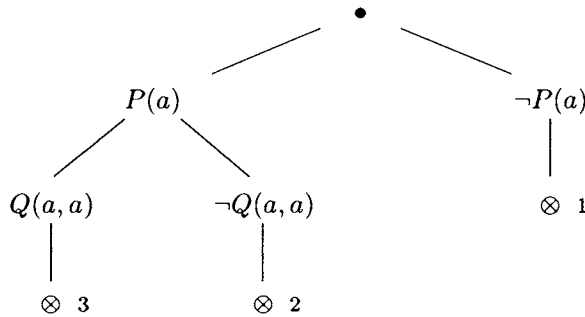
Determine a refutation of the following sets by means of semantic trees:

- (a) $S = \{\{P(x)\}, \{\neg P(x), Q(x, a)\}, \{\neg Q(y, a)\}\}$
 (b) $S' = \{\{P(x), Q(f(x))\}, \{\neg P(a), R(x, y)\}, \{\neg R(a, x)\}, \{\neg Q(f(a))\}\}$

Solution:

- (a) $S = \{\underbrace{\{P(x)\}}_1, \underbrace{\{\neg P(x), Q(x, a)\}}_2, \underbrace{\{\neg Q(y, a)\}}_3\}$.

The Herbrand Universe for S is $H = \{a\}$. Then



2.12.28

Construct the semantic trees and determine the non-satisfiable ground instances corresponding to the following formulae:

- (a) $\sigma : (\exists x) (\forall y) P(x, y) \wedge (\forall y) (\exists x) \neg P(y, x)$
 (b) $\varphi : (\exists x) (\forall y) (\forall z) (\exists w) [(P(x, y) \wedge \neg P(y, x))]$
 (c) $\tau : (\exists x) (\forall y) (\forall z) (\exists w) [P(x, y) \wedge \neg P(z, w)]$

Solution:

(a) We determine the SNF of σ :

$$\begin{aligned}\sigma &\leftrightarrow (\exists x) (\forall y) P(x, y) \wedge (\forall z) (\exists w) \neg P(z, w) \\ &\quad \text{(renaming of bound variables)} \\ &\leftrightarrow (\exists x) (\forall y) (\forall z) (\exists w) [P(x, y) \wedge \neg P(z, w)] \quad \text{(why?)}\end{aligned}$$

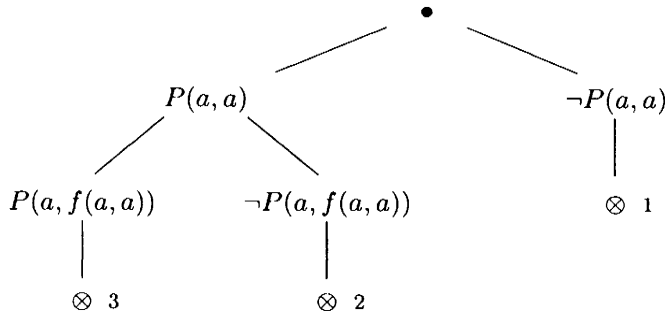
Then

$$\sigma^* : (\forall y) (\forall z) [P(a, y) \wedge \neg P(z, f(y, z))]$$

The Herbrand Universe is

$$H = \{a, f(a, a), f(a, f(a, a)), f(f(a, a), a), f(f(a, a), f(a, a)), \dots\}$$

and the corresponding set of clauses $S = \{\{P(a, y)\}, \{\neg P(z, f(y, z))\}\}$.



One non-satisfiable ground instance of σ is

$$\{\{P(a, f(a, a))\}, \{\neg P(a, f(a, a))\}\}$$

$$\begin{aligned}\text{(b)} \quad \varphi &\leftrightarrow (\exists x) (\forall y) (\forall z) (\exists w) [P(x, y) \wedge \neg P(y, x)] \\ &\leftrightarrow (\exists x) (\forall y) [P(x, y) \wedge \neg P(y, x)]\end{aligned}$$

by Theorem 2.3.6. Then $\varphi^* : (\forall y) [P(a, y) \wedge \neg P(y, a)]$. The set-theoretical form of φ is

$$\{\{P(a, y)\}, \{\neg P(y, a)\}\},$$

and the Herbrand universe $H = \{a\}$. $\{\{P(a, a)\}, \{\neg P(a, a)\}\}$ is a non-satisfiable ground instance.

2.12.29

Find out whether the sentence

$$\sigma : (\exists x) [G(x) \vee F(x)] \rightarrow [(\exists x) F(x) \rightarrow (\exists x) \neg G(x)]$$

is logically true or not, and give the corresponding proof or a counterexample.

Solution:

We construct a CST with an $f\sigma$ origin. If this tableau is contradictory, then we apply Definition 2.6.7 and Theorem 2.10.6. If the CST is not contradictory, then a non-contradictory branch will give us an interpretation in which σ will be false.

2.12.30

Determine a non-satisfiable ground instance for the following sets of clauses:

- (a) $S_1 = \{\{P(x, a, g(x, b))\}, \{\neg P(f(y), z, g(f(a), b))\}\}$
- (b) $S_2 = \{\{P(x)\}, \{\neg P(x), Q(f(x))\}, \{\neg Q(f(a))\}\}$
- (c) $S_3 = \{\{P(x)\}, \{Q(x(f(x))), \neg P(x)\}, \{\neg Q(g(y), z)\}\}$

Solution:

Determine the corresponding Herbrand universe and then construct the corresponding semantic tree.

2.12.31

Let us create a PrL language \mathcal{L} for the study of Euclidean geometry. The special symbols of \mathcal{L} are:

- $P(x)$ predicate symbol of arity 1 interpreted as “ x is a point”
- $E(y)$ predicate symbol of arity 1 interpreted as “ y is a straight line”
- $I(x, y)$ predicate symbol of arity 2 interpreted as “ x belongs to y ”

- (a) What form does the interpretation of \mathcal{L} take? Describe the universe of the interpretation as well as the interpretations of P , E and I ; $\varepsilon(P)$, $\varepsilon(E)$ and $\varepsilon(I)$ respectively.

(b) If

$$\sigma : (\forall x) [P(x) \rightarrow (\exists y) (E(y) \wedge I(x, y))]$$

is a sentence of \mathcal{L} , formulate the intuitive interpretation and examine the truth of σ .

(c) A predicate symbol of arity 2 is added to \mathcal{L} , $\Pi(x, y)$, with an intuitive interpretation “straight lines x and y are parallel”. Describe an interpretation of language $\mathcal{L} \cup \{\Pi\}$. Give a formal presentation of the axioms of $\mathcal{L} \cup \{\Pi\}$.

Solution:

The basic axioms relative to points and straight lines of the plane are:

- There exists at least one point which does not belong to a given straight line.
- Two points belong to exactly one straight line.
- There exist at least two points on every straight line.

From these axioms we can prove, e.g., theorem:

- There exist at least two straight lines passing through a point.

The parallelism of two straight lines is an introduced concept:

- Two straight lines of the plane without any common point are called parallel. (Parallelism *stricto sensu* as opposed to parallelism *lato sensu*, where two straight lines are called parallel if they coincide or have no common points).

(a) An interpretation of \mathcal{L} takes the form $\mathcal{A} = (A, \varepsilon(P), \varepsilon(E), \varepsilon(I))$, where

$$A = \{x \mid x \text{ point of the plane}\} \cup \{x \mid x \text{ straight line of the plane}\},$$

$$\varepsilon(P) = \{x \mid x \text{ is a point}\}, \quad \varepsilon(E) = \{x \mid x \text{ is a straight line}\},$$

and

$$\varepsilon(I) = \{(x, y) \mid x \text{ a point, } y \text{ a straight line, and } x \text{ on } y\}$$

(b) By Definition 2.5.5,

$$\mathcal{A} \models (\forall x) [P(x) \rightarrow (\exists y) (E(y) \wedge I(x, y))] \quad (1)$$

$$\Leftrightarrow \text{for every } c \in A, \mathcal{A} \models P(c) \rightarrow (\exists y) (E(y) \wedge I(c, y))$$

$$\Leftrightarrow \text{for every point } c \text{ and every straight line } y,$$

$$\mathcal{A} \models P(c) \rightarrow (\exists y) (E(y) \wedge I(c, y))$$

$$\Leftrightarrow \text{for every point } c, \mathcal{A} \models P(c) \rightarrow (\exists y) (E(y) \wedge I(c, y))$$

$$\text{AND for every straight line } c, \mathcal{A} \models P(c) \rightarrow (\exists y) (E(y) \wedge I(c, y))$$

But for every straight line c ,

$$\mathcal{A} \models P(c) \rightarrow (\exists y) (E(y) \wedge I(c, y))$$

as for example in the proof of Corollary 1.5.4 and the comments on that proof.

Then by tautology $A \wedge \text{truth} \leftrightarrow A$, we have

$$(1) \Leftrightarrow \text{for every point } c, \mathcal{A} \models P(c) \rightarrow (\exists y) (E(y) \wedge I(c, y))$$

$$\Leftrightarrow \text{for every point } c, \mathcal{A} \models (\exists y) (E(y) \wedge I(c, y)) \text{ or } \mathcal{A} \not\models P(c)$$

$$\Leftrightarrow \text{for every point } c, \text{ there exists some } c' \in A \text{ such that}$$

$$\mathcal{A} \models E(c') \wedge I(c, c') \quad (\text{by the tautology } A \vee (B \wedge \neg B) \leftrightarrow A.)$$

This last sentence, however, results from the axioms:

Let c_1 be a straight line in A . Then c_1 has at least two points c_2 and c_3 . Take c' to be the straight line passing through c and, for example, c_2 .

The sentence σ is then true in \mathcal{A} .

(c) The predicate $\Pi(x, y)$ can be defined in \mathcal{L} . Formally, in $\mathcal{L} \cup \{\Pi\}$ the axioms of the Euclidean plane are:

$$(\forall x) (\exists y) [E(x) \wedge P(y) \wedge \neg I(x, y)]$$

$$(\forall x) (\forall y) (\exists z) \{P(x) \wedge P(y) \wedge E(z) \wedge \neg(x = y)$$

$$\rightarrow [I(x, z) \wedge I(y, z) \wedge ((\forall w) (E(w) \wedge I(x, w) \wedge I(y, w))) \rightarrow (z = w)]\}$$

$$, (\forall x) (\exists y) (\exists z) [E(x) \wedge \neg(y = z) \wedge P(y) \wedge P(z) \rightarrow I(y, x) \wedge I(z, x)]$$

$$\Pi(x, y) \leftrightarrow \neg(\exists z) [P(z) \wedge E(x) \wedge E(y) \wedge I(z, x) \wedge I(z, y)]$$

The last axiom constitutes the definition of predicate Π .

The theorem in the beginning of the solution has the following form:

$$(\forall x) (\exists z) (\exists w) [P(x) \wedge E(z) \wedge E(w) \rightarrow I(x, z) \wedge I(x, w) \wedge \neg(z = w)]$$

By means of the predicate Π we can, for instance, formulate the transitive and symmetric properties of parallelism:

$$(\forall x) (\forall y) (\forall z) [\Pi(x, y) \wedge \Pi(y, z) \rightarrow \Pi(x, z)]$$

and

$$(\forall x) (\forall y) [\Pi(x, y) \leftrightarrow \Pi(y, x)].$$

(Prove that the last two formulae are true in interpretation \mathcal{A}' of $\mathcal{L} \cup \{\Pi\}$, where

$$\mathcal{A}' = (A, \varepsilon(P), \varepsilon(E), \varepsilon(I), \varepsilon(\Pi))$$

and $\varepsilon(\Pi) = \{(x, y) \mid x, y \text{ straight lines, and } x \text{ parallel to } y\}$.)

2.12.32

Find out whether the following sets of clauses are unifiable. If they are, determine the most general unifier (MGU).

- (a) $S = \{\{P(f(a), g(x))\}, \{P(y, y)\}\}$
- (b) $S = \{\{P(a, x, h(g(z)))\}, \{P(z, h(y), h(y))\}\}$
- (c) $S = \{\{Q(f(w), a, z)\}, \{Q(w, b, f(z))\}\}$
- (d) $S = \{\{\text{loves}(w, f(y))\}, \{\text{loves}(\text{George}, \text{football})\}\}$
- (e) $S = \{\{R(w, y), Q(w, f(z), z), \neg R(w, w)\}, \{R(w, z), \neg Q(f(w), w, z)\}\}$
- (f) $S = \{\{P(f(x), a)\}, \{P(y, f(w))\}\}$
- (g) $S = \{\{P(f(x), z)\}, \{P(y, a)\}\}$

Solution:

We apply the unification algorithm:

- (a) $DS_1 = \{f(a), y\}$
 We impose $\theta_1 = \{y/f(a)\}$
 $S_1 = \{\{P(f(a), g(x))\}, \{P(y, y)\}\}$
 $DS_2 = \{g(x), f(a)\}$

We can see that DS_2 has no variables as elements, x occurs only in $g(x)$.

Thus S is not unifiable.

- (b) $DS_1 = \{a, z\}$
 We impose $\theta_1 = \{z/a\}$
 $S_1 = \{\{P(a, x, h(g(a)))\}, \{P(a, h(y), h(y))\}\}$
 $DS_2 = \{x, h(y)\}$
 We impose $\theta_2 = \{x/h(y)\}$
 $S_2 = \{\{P(a, h(y), h(g(a)))\}, \{P(a, h(y), h(y))\}\}$
 $DS_3 = \{g(a), y\}$
 We impose $\theta_3 = \{y/g(a)\}$
 $S_3 = \{\{P(a, h(g(a)), h(g(a)))\}, \{P(a, h(g(a)), h(g(a)))\}\}$

S is hence unifiable, and its MGU is:

$$MGU = \theta_1\theta_2\theta_3 = \{z/a, x/h(y), y/g(a)\}$$

2.12.33

Determine whether the following clauses are unifiable. If they are, give the MGU:

- (a) $S = \{\{Q(a)\}, \{Q(b)\}\}$
 (b) $S = \{\{Q(a, x)\}, \{Q(a, a)\}\}$
 (c) $S = \{\{Q(a, x, f(x))\}, \{Q(a, y, y)\}\}$
 (d) $S = \{\{Q(x, y, z)\}, \{Q(u, h(v, v), u)\}\}$
 (e) $S = \{\{P(x, g(x), y, h(x, y), z, f(x, y, z))\}, \{P(u, v, e(v), w, k(v, w), s)\}\}$

where a, b are constants and f, g, h, e, k functions.

2.12.34

Assume a general unifier θ of a set of clauses S . Prove that:

θ an MGU of S and $\theta\theta = \theta \iff$ for every ω , ω a unifier of S , $\omega = \theta\omega$ holds

Solution:

If θ is an MGU, then for every unifier ω there is a substitution γ , such that:

$$\omega = \theta\gamma = (\theta\theta)\gamma = \theta(\theta\gamma) = \theta\omega$$

Conversely, if ω is a unifier and $\omega = \theta\omega$, then for some substitution γ ,

$$\theta\gamma = \theta(\theta\gamma) = \theta\theta\gamma \quad (1)$$

will hold. Assume $\theta \neq \theta\theta$. Then there exists a variable x and a term q such that $x/y \in \theta$ and $x/q \notin \theta\theta$, which means that $(x/y)\gamma \in \theta\gamma$ and $(x/y)\gamma \notin \theta\theta\gamma$, which gives a contradiction because of (1).

2.12.35

Determine a non-satisfiable ground instance of the set of clauses:

$$S = \{\{P(x, a, g(x, b))\}, \{\neg P(f(y), z, g(f(a), b))\}\}$$

Solution:

We note that we could apply resolution (and determine under which conditions S is non-satisfiable) if x were to be $f(y)$, z were to be a , and x were to be $f(a)$. We apply the substitution:

$$\theta = \{z/u, x/f(a), y/a\}$$

S then becomes:

$$S' = \{\{P(f(a), a, g(f(a), b))\}, \{\neg P(f(a), a, g(f(a), b))\}\}$$

S' is obviously the non-satisfiable ground instance we are seeking (why?).

2.12.36

Let us suppose that:

- (a) There exists a dragon.
- (b) The dragon sleeps in his cave or hunts in the wood.
- (c) If the dragon is hungry, then he cannot sleep.
- (d) If the dragon is tired, then he cannot hunt.

Apply resolution in order to answer the following questions:

- (i) What does the dragon do when he is hungry?
- (ii) What does the dragon do when he is tired?
- (iii) What does the dragon do when he is hungry and tired?

Solution:

We introduce predicates:

Dragon(x) :	x is a dragon	Can(x, y, z) :	y can x in z
Does(x, y, z) :	y does x in z	Hungry(x) :	x is hungry
Tired(x) :	x is tired		

We also assume:

$$\text{Can}(x, y, z) \leftarrow \text{Does}(x, y, z). \quad (*)$$

We convert assumptions (a), (b), (c) and (d) into clauses:

- (a) Dragon(A) \leftarrow
- (b) Does(sleeps, A , cave), Does(hunts, A , wood) \leftarrow
- (c) $\neg\text{Can}(\text{sleeps}, A, \text{cave}) \leftarrow \text{Hungry}(A)$
- (d) $\neg\text{Can}(\text{hunts}, A, \text{wood}) \leftarrow \text{Tired}(A)$

We then convert assumptions (a) to (d) and (*) into set-theoretical forms:

- (1) {Dragon(A)}
- (2) {Does(sleeps, A , cave), Does(hunts, A , wood)}
- (3) { $\neg\text{Hungry}(A)$, $\neg\text{Can}(\text{sleeps}, A, \text{cave})$ }
- (4) { $\neg\text{Tired}(A)$, $\neg\text{Can}(\text{hunts}, A, \text{wood})$ }

(5) $\{\text{Can}(x, y, z), \neg\text{Does}(x, y, z)\}$

(i) We add the following clauses to clauses (1) to (5).

(6) $\{\text{Hungry}(A)\}$

(7) $\{\neg\text{Does}(x, y, z)\}$

Clause (7) constitutes the goal of the resolution. We will try to conclude \square , by adequately instantiating variables x , y and z .

(8) $\{\text{Does}(\text{sleeps}, A, \text{cave})\}$, $x = \text{sleeps}$, $y = A$, $z = \text{cave}$ by (2) and (7)

(9) $\{\text{Can}(\text{sleeps}, A, \text{cave})\}$ by (5) and (8)

(10) $\{\neg\text{Can}(\text{sleeps}, A, \text{cave})\}$ by (4) and (6)

(11) \square by (9) and (10)

Then when the dragon is hungry, it hunts in the wood.

(ii) We add the following clauses to clauses (1) to (5).

(6) $\{\text{Tired}(A)\}$

(7) $\{\neg\text{Does}(x, y, z)\}$ then

(8) $\{\text{Does}(\text{hunts}, A, \text{wood})\}$, $x = \text{hunts}$, $y = A$, $z = \text{wood}$ by (2) and (7)

(9) $\{\text{Can}(\text{hunts}, A, \text{wood})\}$ by (5) and (8)

(10) \square by (9) and (10)

(iii) We add the following clauses to clauses (1) to (5).

(6) $\{\text{Hungry}(A)\}$

(7) $\{\text{Tired}(A)\}$

(8) $\{\neg\text{Does}(x, y, z)\}$

Clause (8) is the goal of the resolution. Then:

(9) $\{\neg\text{Can}(\text{sleeps}, A, \text{cave})\}$ by (3) and (6)

(10) $\{\neg\text{Can}(\text{hunts}, A, \text{wood})\}$ by (4) and (7)

(11) $\{\neg\text{Does}(\text{sleeps}, A, \text{cave})\}$, $x = \text{sleeps}$, $y = A$, $z = \text{cave}$ by (5) and (9)

(12) $\{\neg\text{Does}(\text{hunts}, A, \text{wood})\}$, $x = \text{hunts}$, $y = A$, $z = \text{wood}$ by (5) and (10)

(13) $\{\text{Does}(\text{hunts}, A, \text{wood})\}$ by (2) and (11)

(14) $\{\text{Can}(\text{hunts}, A, \text{wood})\}$, $x = \text{hunts}$, $y = A$, $z = \text{wood}$ by (5) and (13)

(15) \square by (10) and (14)

Note that the inconsistency of clauses (1) to (8) was proved without the use of clause (8), which was also our goal. According to the data, the situation of a both hungry and tired dragon is contradictory. We thus cannot decide what the dragon will do when it is both hungry and tired.

2.12.37

Assume predicates:

$T(x, y, u, v)$: interpreted as “ $xyuv$ is a trapezium and x, y, u and v are the upper left, upper right, lower right and lower left vertices respectively”.

$P(x, y, v, u)$: interpreted as “the straight line sections xy and vu are parallel”.

$E(x, y, z, u, v, w)$: interpreted as “ $\widehat{xyz} = \widehat{uvw}$ ”.

and sentences:

A_1 : $(\forall x)(\forall y)(\forall u)(\forall v) [T(x, y, u, v) \rightarrow P(x, y, v, u)]$
(definition of a trapezium)

A_2 : $(\forall x)(\forall y)(\forall u)(\forall v) [P(x, y, v, u) \rightarrow E(x, y, v, u, v, y)]$
(if $xy \parallel vu$ then, $\widehat{xyv} = \widehat{uvw}$)

A_3 : $T(a, b, c, d)$ ($abcd$ is a trapezium)

Prove with resolution that A_1, A_2 and A_3 imply $E(a, b, d, c, d, b)$.

Solution:

We just need to prove that the sentence

$$A_1 \wedge A_2 \wedge A_3 \wedge \neg E(a, b, d, c, d, b) \quad (*)$$

is not satisfiable (why?).

The set-theoretical form of $(*)$ is:

$$\begin{aligned} S = \{ & \{ \neg T(x, y, u, v), P(x, y, v, u) \}, \\ & \{ \neg P(x, y, v, u), E(x, y, v, u, v, y) \}, \\ & \{ T(a, b, c, d) \}, \\ & \{ \neg E(a, b, d, c, d, b) \} \} \end{aligned}$$

This gives

- (1) $\{\neg T(x, y, u, v), P(x, y, v, u)\}$
- (2) $\{\neg P(x, y, v, u), E(x, y, v, u, v, y)\}$
- (3) $\{T(a, b, c, d)\}$
- (4) $\{\neg E(a, b, d, c, d, b)\}$
- (5) $\{\neg P(a, b, d, c)\}$ unification and resolution in (2) and (4).
- (6) $\{\neg T(a, b, c, d)\}$ unification and resolution in (1) and (5).
- (7) \square by (3) and (6).

2.12.38

Prove with resolution and CST:

- (a) $\{(\forall x) [A(x) \rightarrow (B(x) \wedge C(x))], (\exists x) [A(x) \wedge D(x)]\} \models (\exists x) [D(x) \wedge C(x)]$
- (b) $\left\{ \begin{array}{l} (\exists x) [P(x) \wedge (\forall y) (T(x, y) \rightarrow Q(x, y))] , \\ (\forall x) [P(x) \rightarrow (\forall y) (W(y) \rightarrow \neg Q(x, y))] \end{array} \right\} \models (\forall x) [T(x) \rightarrow \neg W(x)]$

Solution:

- (a) We construct a CST with an origin $f[(\exists x) [D(x) \wedge C(x)]]$. We then concatenate in a conjunction (line!!!) $t[(\forall x) [A(x) \rightarrow (B(x) \wedge C(x))]]$ and $t[(\exists x) [A(x) \wedge D(x)]]$.

For resolution, we convert into an SNF the conjunction

$$\begin{aligned} & ((\exists x) [A(x) \wedge D(x)]) \wedge ((\forall x) [A(x) \rightarrow (B(x) \wedge C(x))]) \\ & \wedge (\neg(\exists x) [D(x) \wedge C(x)]) \end{aligned}$$

(from the commutativity of \wedge , we first took the formula with the existential quantifier in order to have simpler terms in the corresponding SNF.) We then continue with the set-theoretical form of (a).

III Logic Programming:

The PROLOG Paradigm

Τὸ ὑφεισθηκὸς δεῖ τέλος ἐπιλογίζεσθαι καὶ
πᾶσαν τὴν ἐνἀργειαν, ἐφ' ἣν τὰ δοξαζόμενα
ἀνάγομεν· εἰ δὲ μὴ, πάντα ἀχρισίας καὶ
ταραχῆς ἔσται μεστά.

We must take into account the end sought and
all clear evidence of sense to which we refer our
opinions. For otherwise everything will be full
of uncertainty and confusion.

Epicurus

3.1 PROLOG and Logic Programming

3.1.1 Introduction

In the previous chapters, within the context of Propositional Logic and Predicate Logic, we examined and analyzed the notions which are critical to the study of Logic Programming. Our goal now is to present *Logic Programming* as a direct application of all the elements of Mathematical Logic that we mentioned in the two previous chapters, emphasizing wherever possible, the direct relationship between Mathematical Logic and the programming language PROLOG.

The term *Programming Language* does not exactly correspond to reality. PROLOG, like FORTRAN and PASCAL, is not a language; it is only a notation [DiSc90] by means of which we represent and formalize data and procedures. The so called “programming languages” differ greatly from the natural languages in terms of their expressive power, their usage, and their interpretation. We will continue utilizing the term *Programming Language*, because it is an established term, but we will always be referring to the set of expressions which are present in every kind of programming.

The principal role of Propositional Logic and Predicate Logic is the determination of mathematical models, which make possible the proof of truth or falsehood of a proposition derived from a set of other propositions and assumptions. Within this context, we have already dealt with models which describe the rule of modus ponens, with proofs using tableaux, and with proofs by resolution. This last method is the strongest link between Mathematical Logic and Logic Programming. At the same time, it somehow bridges the gap between the abstract mathematical models and the operation of a programming language such as PROLOG.

In the first chapter we studied the resolution as a proof procedure of Propositional Logic. Furthermore, in the second chapter, by studying the concepts of canonical forms, Skolem forms, and the notions of substitution and unification, we extended resolution into a proof procedure in Predicate Logic as well. Now we will define a programming language, by means of which we will represent and elaborate on the everyday practical experience and events of our world, using as many elements from the science of mathematics as we can.

The dominant characteristics of this language are the formalization of the given information, the operation of the inference mechanism, as well as the relationship between this mechanism and Mathematical Logic.

The representation is formally defined by Horn clauses, which not only describe in simple terms the events of our world, but can also be transformed easily into Skolem Normal Forms. The inference mechanism of the language, with substitution, unification and resolution, offers an algorithmic mathematical tool for data management and derivation of conclusions from a set of Horn clauses.

We know that resolution examines the data of a problem and guarantees that if there exists a solution, this solution will be found. Having this in mind, we are in a position to make a first intuitive presentation of PROLOG by means of an example.

We assume the following data expressed as Horn clauses of Predicate Logic:

- p_1 $\text{Person}(\text{Marcus}) \leftarrow$
- p_2 $\text{Person}(\text{Aurelius}) \leftarrow$
- p_3 $\text{Mortal}(x_1) \leftarrow \text{Person}(x_1)$
- p_4 $\text{Born_Pompei}(\text{Aurelius}) \leftarrow$
- p_5 $\text{Born_Pompei}(\text{Marcus}) \leftarrow$
- p_6 $\text{Born}(\text{Aurelius}, 45) \leftarrow$
- p_7 $\text{Born}(\text{Marcus}, 40) \leftarrow$
- p_8 $\text{Died}(x_2, 79) \leftarrow \text{Born_Pompei}(x_2)$
- p_9 $\text{Erupted}(\text{volcano}, 79) \leftarrow$
- p_{10} $\text{Dead}(x_3, t_2) \leftarrow \text{Mortal}(x_3), \text{Born}(x_3, t_1), t_2 - t_1 > 150$
- p_{11} $\text{Not_Alive}(x_4, t_3) \leftarrow \text{Dead}(x_4, t_3)$
- p_{12} $\text{Dead}(x_5, t_4) \leftarrow \text{Not_Alive}(x_5, t_4)$
- p_{13} $\text{Dead}(x_6, t_6) \leftarrow \text{Died}(x_6, t_5), t_6 > t_5$
- p_{14} $\leftarrow \text{Not_Alive}(x, 1987)$

where for uniformity with the PROLOG formalism which will follow, we write the indices beside the variables and terms; for example, x_1 becomes $x1$.

The first thirteen Horn clauses represent the information which we give the program. p_{14} is the query we make:

Who is not alive in 1987?

The corresponding set-theoretic form of the sentences p_1, \dots, p_{14} is:

- (1) $\{\text{Person}(\text{Marcus})\}$
- (2) $\{\text{Person}(\text{Aurelius})\}$
- (3) $\{\text{Mortal}(x_1), \neg \text{Person}(x_1)\}$
- (4) $\{\text{Born_Pompei}(\text{Aurelius})\}$
- (5) $\{\text{Born_Pompei}(\text{Marcus})\}$
- (6) $\{\text{Born}(\text{Aurelius}, 45)\}$
- (7) $\{\text{Born}(\text{Marcus}, 40)\}$
- (8) $\{\text{Died}(x_2, 79), \neg \text{Born_Pompei}(x_2)\}$
- (9) $\{\text{Erupted}(\text{volcano}, 79)\}$
- (10) $\{\text{Dead}(x_3, t_2), \neg \text{Mortal}(x_3), \neg \text{Born}(x_3, t_1), \neg(t_2 - t_1 > 150)\}$
- (11) $\{\text{Not_Alive}(x_4, t_3), \neg \text{Dead}(x_4, t_3)\}$
- (12) $\{\text{Dead}(x_5, t_4), \neg \text{Not_Alive}(x_5, t_4)\}$
- (13) $\{\text{Dead}(x_6, t_6), \neg \text{Died}(x_5, t_5), \neg(t_6 > t_5)\}$
- (14) $\{\neg \text{Not_Alive}(x, 1987)\}$

The corresponding representation in PROLOG is:

- (1) `person(marcus).`
- (2) `person(aurelius).`
- (3) `mortal(X1):- person(X1).`
- (4) `born_Pompei(aurelius).`
- (5) `born_Pompei(marcus).`
- (6) `born(aurelius,45).`
- (7) `born(marcus,40).`
- (8) `died(X2,79):- born_Pompei(X2).`
- (9) `erupted(volcano,79).`
- (10) `dead(X3,T2):- mortal(X3), born(X3,T1), gt(T2,150).`
- (11) `not_Alive(X4,T3):- dead(X4,T3).`
- (12) `dead(X5,T4):- not_Alive(X5,T4).`
- (13) `dead(X6,T6):- died(X5,T5), gt(T6,T5).`
- (14) `? not_Alive(X,1987).`

where `gt(X,Y)` means $X > Y$. We use the PROLOG notation, writing constants with lower case letters while variables are written in upper case letters. The period symbol “.” is used in most PROLOG versions to denote the end of a Horn clause input.

PROLOG, similarly to Predicate Logic, tries to prove the goal. In its answer, if the goal is valid, PROLOG will instantiate; that is, it will give all possible values of the variables which validate the goal. Let us examine how this happens and how it relates to substitution, unification and resolution.

Step 1 :

Our goal is:

`Not_Alive(x,1987)`

PROLOG looks through the data for a Horn clause such that its head can be unified with the goal. In fact, the substitution $\theta_1 = \{x4/x, t3/1987\}$ unifies the goal with the head of the formula (11). PROLOG immediately tries to verify the body of the formula (11), by setting as a new goal, one after the other, all the assumptions of this formula.

The resolution of (14) and (11) gives:

- (15) `{¬Dead(x,1987)}` from (11) and (14).

Step 2:

Our goal is now:

$$\text{Dead}(x, 1987)$$

PROLOG, just like resolution does, now tries to unify the goal “Dead(x , 1987)” with a head of some clause from the data. This is achieved through the substitution $\theta_2 = \{x3/x, t2/1987\}$ in formula (10).

Step 3:

The new goal for PROLOG is the triplet of assumptions from (10), i.e.:

$$\text{Mortal}(x), \quad \text{Born}(x, t1) \quad \text{and} \quad \text{gt}(1987 - t1, 150)$$

which it tries to satisfy in the order that they appear in rule (10). The resolution method respectively, would give us rule (16):

$$(16) \quad \{\neg \text{Mortal}(x), \neg \text{Born}(x, t1), \neg \text{gt}(1987 - t1, 150)\}$$

Step 4:

PROLOG unifies “Mortal(x)” with the head of sentence (3) by using the substitution $\theta_3 = \{x1/x\}$. Likewise, resolution would use the same substitution to unify “Mortal(x)” with “Mortal($x1$)” of (3).

Step 5:

Our new goal is now:

$$\text{Person}(x)$$

which in turn is unified through the substitution $\theta_4 = \{x/\text{Marcus}\}$ with the fact described by formula (1).

Using the resolution method we would have:

$$(17) \quad \{\neg \text{Person}(x), \neg \text{Born}(x, t1), \neg \text{gt}(1987 - t1, 150)\} \quad \text{from (3) and (16).}$$

$$(18) \quad \{\neg \text{Born}(\text{Marcus}, t1), \neg \text{gt}(1987 - t1, 150)\} \quad \text{from (1) and (17).}$$

Step 6:

PROLOG continues its proof procedure and tries to satisfy the next goal from the triplet of goals presented in step (3), which now becomes:

$$\text{Born}(\text{Marcus}, t1)$$

The unification succeeds through the substitution $\theta_5 = \{t1/40\}$ with the head of rule (7).

The resolution, correspondingly, would conclude the formula

$$(19) \quad \{\neg \text{gt}(1987 - 40, 150)\} \quad \text{from (7) and (18).}$$

Step 7:

The last goal to be satisfied from the triplet of goals from step (3) is

$$\text{gt}(1987 - 40, 150)$$

which is satisfied by PROLOG since $1987 - 40 > 150$.

Step 8:

The goal of step (2), “Dead(x , 1987)”, has been satisfied with the instantiation $x = \text{Marcus}$. The goal of the first step, “Not_Alive(x , 1987)”, is therefore satisfied with the same instantiation.

In this step, resolution deduces clause

$$(20) \quad \square \quad \text{from (19).}$$

Step 9 . . . :

PROLOG does not stop here. It goes on, trying to unify the most recently selected goal in a different manner, in order to find all possible solutions. Once the last goal has been satisfied in every possible way and all the valid solutions have been produced, the procedure is repeated for the immediately preceding

goal. When this one is satisfied in some different fashion, the program goes on by satisfying again the initial “last” goal using the substitutions which were made during the one before last goal, and continues along these lines: the final goal “gt(1987 – 40, 150)”, cannot be unified in any other way, therefore PROLOG tries to unify, using a different substitution this time, the immediately preceding goal “Born(Marcus, 40)”. With this method it eventually unifies “Mortal(x)” by means of the substitution $\theta = \{x/\text{Aurelius}\}$ and with fact (2). It produces, just like in step (6), but for $x = \text{Aurelius}$ this time, the new solution:

Not_alive(Aurelius, 1987)

This slick mechanism of PROLOG which finds all the possible solutions is called **backtracking** and will be analyzed in detail in its respective section.

3.1.2 Logic and Programming

The difference that distinguishes Logic Programming and the PROLOG language from traditional programming and languages like FORTRAN, BASIC, COBOL, and PASCAL, etc., lies in the fundamental principles of logic programming; both in the design and the implementation of a logic program.

A program consists of two building elements, **Logic** and **Control** [Kowa79, Lloy87]. By the term “Logic” we denote all those principal notions that determine WHAT a program is doing, whereas by the term “Control” we mean all those syntactic notions that determine HOW it is doing it (for example, the algorithm which solves a problem). If we wished to describe this using a single equation, we would write:

$$\text{PROGRAM} = \text{LOGIC} + \text{CONTROL}$$

A traditional program written in BASIC or in some other traditional programming language consists of **commands** which describe the actions which have to be executed step by step by the computer in order for the program to produce the desired result. For example, the command in a BASIC program

```
10      LET  X = X + 5
```

increases by 5 the content of the memory address which corresponds to the variable X .

Programming languages like BASIC are characterized by **imperative** commands which describe the **step by step** behaviour of the program, so that, after a finite sequence of those commands, the correct and expected result is produced. The structure of these commands during the design of the program comprises the element of CONTROL.

Likewise, the element of LOGIC in the above command is the expression “ $X+5$ ”, which is not a command by itself, but only a small **descriptive** program. This program directly describes the arithmetic value which has to be computed and indirectly only how this value will be computed.

Therefore, BASIC is an **imperative** language which possesses a **descriptive** element.

On the contrary, a PROLOG program could contain the sentence:

```
nice(X):- loves(X,people)
```

which is simply a logical definition of the relationship between predicates “nice” and “loves”.

The design, therefore, of a PROLOG program is based upon the correct selection of predicates which define the relations between objects, such that the connection between the input data and the information that we expect as output is fully defined.

In general, a program in a traditional programming language expresses a function from the input to the output of the program, whereas a program in a Logic Programming language expresses a *relation between the data* [Watt90]. As relations are more general than functions (functions are necessarily relations, but relations are not in general functions), Logic Programming has greater capabilities than traditional programming.

The selection of the predicates, and the relations which are expressed by these predicates, constitute the logic element in PROLOG. The control is found not only in the order in which we arrange the clauses, but also in a number of its structural control elements, like the cut “!” (section 4.3), which can be used as syntactic objects of PROLOG. PROLOG is therefore a **descriptive** language which contains an **imperative** structural element as a control element.

Let us take as an example a program which reads two numbers and prints the greater of them. In order to make the difference between traditional programming and Logic Programming more concise, we will first give the program in BASIC and then exactly the same program in PROLOG:

Program in BASIC

```

10      INPUT "NUMBER1", X
20      INPUT "NUMBER2", Y
30      IF X > Y THEN 60
40      PRINT Y
50      GOTO 70
60      PRINT X
70      END

```

Program in PROLOG

```

program:- write("NUMBER1"), real(X), nl,
          write("NUMBER2"), real(Y), nl,
          greater(X,Y,Z), write(Z).
greater(X,X,X).
greater(X,Y,Y):- X < Y.
greater(X,Y,X):- Y < X.
? program.

```

“real”, subsection 3.5.4, is a special predicate of PROLOG which checks whether a number is real. Using “nl”, every number is written in a different output line.

The program in BASIC is just a sequence of commands. These commands, which are executed in the order that the program indicates, constitute the control, i.e., the design and the flow of the program. The logic element in the BASIC program is found in the relation “<”.

On the contrary, the PROLOG program is a collection of clauses which fully describes the relation which completely determines the order of two numbers, i.e., the predicate “greater”. This collection of clauses expresses the Logic, which also plays the dominant role in a PROLOG program, whereas the Control is found in the order in which we arrange and define the predicates.

A typical program consists of the data we want to process, and the sequence of actions we want to perform. Once we formulate the facts of the problem and the procedures which will give us the results, the control structure specifies the order in which the different procedures must be executed. In other words, we have a clearly defined sequence of procedures, some of which are repetitive (e.g., loops). The selection of the formalism which we will use is very important, because the automation of processing depends on it. The programmer is totally responsible for this selection. Therefore, a fundamental element of a typical program is the **flow**, that is, the order of the procedures according to which they will be carried out.

3.1.3 *Logic Programming*

One of the major drawbacks of programs in traditional programming is that they require permanent modification and adaptation to new demands. However, any partial modification in a classic program, usually affects its overall flow. For this reason checking an even slightly modified program is a demanding and time-consuming task.

One solution to the problems of traditional programming was provided by Logic Programming through the language of Predicate Logic. A problem which is to be solved using the help of the computer is introduced as a collection of Predicate Logic clauses. Some of these clauses express facts; for instance, in the example of subsection 3.1.2, the clause “**greater**(X, X, X)”. Others express rules for dealing with facts, as in the clause “**greater**(X, Y, X):- $X > Y$ ” from the same example, and still others express the queries which have to be answered giving the solution to the problem. Thus in Logic Programming, and specifically in PROLOG, after applying the rules, the solution to the problem will either be an answer to the queries of the form **yes** or **no**, or a **set of values** which will form the desired solution.

The commands therefore in Logic Programming are not of an exclusively functional nature, as in classical programming; but are predicates, elements of the Predicate Logic language, which are either true or false according to the interpretation of their terms. The truth or falsehood of certain predicates according to the interpretations of the variables provides the precisely corresponding answer of the form “**yes**” or “**no**” to the questions of the program.

The answers to the queries in the program are given ONLY in relation to the information which we have given the program. Therefore, if we ask the program in the example of subsection 3.1.1 if 4 is a perfect square, that is, the square of an integer, the program, not having been given the predicate which characterizes the perfect square, will answer “no”. This means that the goal:

? perfect_square(4).

fails, Remark 1.9.9. We have to pay attention here. The failure of PROLOG to verify the goal does not mean that the goal is really incorrect, it only means that using the facts of the program and the inference mechanism that we have, we cannot conclude that this specific goal is true. This is the so called **Closed World Assumption** (see also subsection 3.6.1), [Watt90]:

A predicate Q of Predicate Logic is assumed false by the program if the program cannot prove that Q is true.

Therefore, every PROLOG program appears as a complete description of the universe, and the universe is fully characterized by the data of the program. Whatever is important in the universe is described by the data, and the relevant goals succeed; whereas whatever is not specified in the program is not known, and the relevant goals fail.

3.1.4 *Historical Evolution*

After Herbrand’s algorithm (1930), using which we can find an interpretation which contradicts a formula φ of Predicate Logic if φ is not logically true, Gilmore (1960) was the first to try to implement Herbrand’s method on a computer [ChLe73]. Gilmore’s implementation was based on the fact that a clause is logically correct, Definition 2.5.19, if and only if its negation is contradicted in some interpretation of the language. Consequently, in his program, a procedure exists which forms PrL sentences which are periodically checked for inconsistency, Definition 2.5.16. However, the program of Gilmore was not capable of analyzing very complex PrL formulae.

After the publication of Robinson’s [Robi65] unification algorithm, Loveland (1970) was the first to use linear resolution with a selection function. This is the resolution during which we pick a clause, we resolve it with another, we resolve the resolvent (Definition 1.9.11) with a third clause and we continue always resolving the last resolvent until we find an empty programming clause [ChLe73, Lloy87].

The first steps (1972) in Logic Programming are attributed to Kowalski and Colmerauer [StSh86]. First Kowalski expressed the procedural interpretation of the Horn clause logic and showed that a rule of the form:

$$A \text{ :- } B_1, B_2, \dots, B_n$$

can be both read and executed by a recursive programming language.

At the same time Colmerauer and his research team developed in FORTRAN a programming language to be used as a theorem prover, which implemented the procedural interpretation of Kowalski. The founding stone of PROLOG (PROgrammer en LOGique) was set!

The first PROLOG interpreter, i.e., the program which translates a text understandable by people (source file) to a language understandable by the machine (machine code), was written by Roussel in Marseilles in 1972 using ALGOL, and was based on the theoretical works of Colmerauer.

However the question of whether Logic, and consequently Mathematics, can be used as a programming language, must be answered negatively [Watt90]:

A problem expressed in a programming language, has to be solvable by a computer, whereas in Logic and Mathematics there are problems which do not have algorithmic solutions, and hence the computer is not able to solve them.

One example of this is the decision problem in Predicate Logic, section 2.11.

Hence Logic Programming, which means the languages which are based on Logic, is always restricted to a part of logic formulae; for instance PROLOG only deals with Horn clauses, i.e., a subset of the PrL sentences.

Many researchers have recently focused in the relations between several programming languages; such as Functional Programming, i.e., programming based on λ -calculus [Thay88], Logic Programming, and Object-Oriented Programming, i.e., programming based on the theory of types, according to which every object is represented by a set of properties, values and procedures [Thay88]. A lot of effort has been put into finding suitable transformations and equivalences that enable the transition from one programming language to others, as well as the use of different formalisms in one program at the same time.

Most of the Logic Programming languages, which in fact are logic theorem proving systems using the resolution method, are the first steps towards optimal Logic Programming.

For optimal programming, however, we have to solve the problem of Control completely, that is:

- (i) Have available more satisfactory and more slick Control procedures in every language of Logic Programming
- (ii) Be able to transfer Control during the design and execution of a program exclusively to the computer.

The solution of the Control Problem will enable future programmers and users to limit their interaction with the computer to the complete and concise definition of the problem. The program in turn will take over the solution of the problem and the total control of the program flow.

3.2 Program Structure

3.2.1 The Program Elements

A PROLOG program consists of data and queries to which the program has to give an answer. The data and queries are Horn clauses, Definition 2.4.4.

The data have one of the following forms:

$$A(c_1, \dots, c_k) \leftarrow \quad (*)$$

or as we symbolically write in PROLOG

$$A(c_1, \dots, c_k) .$$

where A is a predicate and c_1, \dots, c_k are constants of PrL. The period symbol “.” signifies the point where a given formula stops.

$$A(a_1, \dots, a_k) \leftarrow B_1(b_1, \dots, b_\ell), \dots, B_j(d_1, \dots, d_m) \quad (**)$$

or as we symbolically write in PROLOG

$$A(a_1, \dots, a_k) :- B_1(b_1, \dots, b_\ell), \dots, B_j(d_1, \dots, d_m) .$$

where A, B_1, \dots, B_j are predicates and $a_1, \dots, a_k, b_1, \dots, b_\ell, \dots, d_1, \dots, d_m$ are terms (Definition 2.2.1) of Predicate Logic.

Data of the form $(*)$ are the **facts** of the program (Definition 2.4.6).

Data of the form $(**)$ are called the **rules** of the program. The rules express the relations between the predicates occurring in the facts. The rules, which are implicative formulae of Predicate Logic on which the program applies unification and resolution, form the **procedural part** of the program. $A(a_1, \dots, a_k)$ is called the **head**, and $B_1(b_1, \dots, b_1), \dots, B_j(d_1, \dots, d_m)$ is called the **tail** or **body** of the rule.

The facts and the rules form the **database** of the program [Brat90].

The **queries** are **goals** (Definition 2.4.5), i.e., Horn clauses of the form:

$$B_1(b_1, \dots, b_\ell), \dots, B_j(d_1, \dots, d_m)$$

or symbolically in PROLOG:

$$? B_1(b_1, \dots, b_\ell), \dots, B_j(d_1, \dots, d_m).$$

where B_1, \dots, B_j are predicates and $a_1, \dots, a_k, b_1, \dots, b_\ell, \dots, d_1, \dots, d_m$ terms (Definition 2.2.1) of Predicate Logic.

The period “.” following each fact, rule or query denotes the end of the corresponding Horn clause.

Let us study the PROLOG program of Example 2.4.9 with the query set in Example 2.10.12, “What can Peter steal?”:

```
thief(peter).
likes(mary, food).
likes(mary, wine).
likes(peter, money).
likes(peter, X) :- likes(X, wine).
can_steal(X, Y) :- thief(X), likes(X, Y).
? can_steal(peter, Y).
```

The facts are the first four Horn clauses, the next two clauses are rules, and the last clause is the goal (the query) of the program. PROLOG will respond with

```
Y = money
Y = mary
```

which means that Peter can steal money and Mary, something which we already knew from Example 2.10.12.

Hence a PROLOG program is a collection of facts, rules and queries, without any special directions for the flow and control of the program. This structure exhibits the great slickness of PROLOG: extensions and improvements can be achieved by adding or deleting facts and rules using the commands “**assert**” and “**retract**” which we will study in subsection 3.5.1.

The facts in a PROLOG program can be interpreted as **declarations** regarding the universe of the program, hence they are allegations which describe a specific world. The rules [Xant90] can be interpreted not only as declarations, but also as **procedures** of the program. Classical programming languages essentially allow a procedural interpretation, whereas PROLOG is probably the only programming language [Xant90, Brat90, ClMc84], which allows both a **procedural** and a **declarative interpretation**.

3.2.2 The Facts

As we saw in subsection 3.2.1, a fact is a relation, i.e., a predicate, between a number of objects, which are terms of a PrL language. Thus, a fact is a PrL sentence (Definition 2.2.17). The facts in PROLOG express directly corresponding facts of the spoken language:

Spoken language	PROLOG
the canary is a bird	bird(canary).
John is a man	man(john).
Peter likes Mary	likes(peter,mary).
Peter can steal Mary	can_steal(peter,mary).
Mary's parents are John and Kate	parents(mary,john,kate).

The names of predicates and constants always start with a lower case letter. The words in predicates which are expressed with more than one word, are connected with an underscore, “_”. Periods denote the end of each fact.

The position of the terms in the predicates is ordered. The fact:

likes(mary,peter).

signifies that:

“Mary likes Peter”

and it is different from the fact:

likes(peter,mary).

The selection of predicates and names which will be used is carried out by the programmer. His interpretation of facts is therefore important. For example, the fact “`father(john, chris)`” can be interpreted either as “John is the father of Chris” or as “Chris is the father of John”. We can pick either one of the two options, as long as we do not pick both! We should not expect correct results from a program which contains the facts:

```
father(john, chris).  
father(george, nick).
```

when the first fact is interpreted as:

“the father of John is Chris”

yet the second fact as:

“the father of Nick is George”

3.2.3 *The Rules*

A rule in PROLOG is an implicative PrL formula (subsection 3.2.1). The rules express information and relations more general and more complex than the information expressed by the facts. For example,

```
father(chris, george).  
son(nick, john).  
father(X, Y) :- son(Y, X).
```

The first two clauses are facts. They express the relationship between specific people, Chris—George and Nick—John respectively. The third clause expresses a general inferential rule which has to do with the predicates of the data: if Y is the son of X , then X is the father of Y .

The variables in both the rules and the queries are written in capital letters. The period denotes the end of each rule. The head is separated from the tail with the symbol “:- ” (neck symbol). If the tail of the rule consists of more than one predicate, then these predicates are separated from each other by a comma “ , ”.

Let us look at an example:

```
likes(john,ice_cream).
likes(john,mary).
likes(john,backgammon).
food(ice_cream).
eats(X,Y): food(Y), likes(X,Y).
```

Here the tail of the rule is composed of two predicates,

“food(Y)” and “likes(X,Y)”

This rule conjectures that for all X and Y (Definition 2.4.1), if Y is food and X likes Y , then X eats Y (Definition 2.4.1 (ii)).

In PROLOG we can write two or more rules which share the same head as a new rule, with head the common head and using the symbol “;”. For example, the rules:

```
likes(X,Y):- likes(Y,X).
likes(X,Y):- good(Y).
```

can be written as one rule:

```
likes(X,Y):- likes(Y,X); good(Y).
```

and the meaning of it is “ X likes Y , if Y likes X , or if Y is good”.

PROLOG enumerates the facts and the rules of every database. For example, in the previous database, “likes(john,ice_cream)” is the first Horn clause of the database and “eats(X,Y) : food(Y); likes(X,Y)” is the fifth. As we will see in subsection 3.4.1, the order in which the facts and the rules are entered in the database is very important for the derivation of conclusions.

3.2.4 The Queries

The **queries** or **goals** in PROLOG, are questions related to the conditions and the predicates of Predicate Logic which appear in the data of the program. For example, for the data of subsection 3.2.2 we can ask:

```
(10)   ? likes(gas,X).
        ? likes(X,Y), food(Y).
```

and PROLOG will respond to the first query with

```
X = ice_cream
X = mary
X = backgammon
```

and to the second question with

```
X = john
Y = ice_cream
```

The queries begin with a “?”, question mark, and end with a period. A query, for example

```
? likes(gas, X).
```

can be answered in a lot of different ways. A **complex query**, like

```
? likes(X, Y), food(Y).
```

has more than one subgoal, which are separated from one another with a comma. In the complex queries, the same variables always refer to the same terms. For example, we asked

```
? likes(X, Y), food(Y).
```

hence we asked the program to find X and Y , such that X likes Y and Y is a food. The query

```
? likes(X, Y), food(X).
```

would fail, that is PROLOG would not succeed in finding a suitable value for X , and would therefore answer “no”.

Let us for example examine the description of a party on the facing page.

Queries like (1), (2) and (3), are **data verification queries**, whereas queries similar to queries (4) through (7), are **data management queries**. Queries (3), (6) and (7) are **complex** or **conjunctive queries**.

```

likes(gas,helen).
likes(john,mary).
likes(jane,nick).
likes(kate,nick).
listens(john,rock).
listens(tim,reggae).
listens(anne,heavy_metal).
drinks(gas,vodka).
drinks(john,gin).
drinks(kate,wine).
drinks(jane,wine).
drinks(anne,coke).
smokes(gas).
smokes(kate).
couple(anne,X):- listens(X,rock).
couple(nick,X):- drinks(X,wine), likes(X,nick), smokes(X).
(1)  ? likes(john,mary).
(2)  ? likes(jack,mary).
(3)  ? drinks(jane,wine), listens(jane,rock).
(4)  ? couple(nick,X).
(5)  ? listens(X,Y).
(6)  ? likes(kate,X), likes(jane,X).
(7)  ? likes(X,Y), listens(X,rock), drinks(Y,vodka).

```

PROLOG will respond with the following:

```

(1)  yes
(2)  no
(3)  no
(4)  X = kate
(5)  X = john
      Y = rock
      X = tim
      Y = reggae
      X = anne
      Y = heavy_metal
(6)  X = nick
(7)  no

```

In queries (2), (3) and (7) the PROLOG program failed (Remark 1.9.9 (2)):

For query (2) there is no information concerning Jack and Mary in the data of the program.

For query (3) the first subgoal succeeds, because Jane drinks wine, however, there is no information regarding the type of music the Jane listens to. Hence the second subgoal and consequently (why?) goal (3) fails.

Query (7) fails because in the data for the party there are no values for X and Y which satisfy all three subgoals at the same time.

3.3 Syntax of Data

3.3.1 *The Objects of PROLOG*

As we have already said, the language which we use in PROLOG is a subset of the language in Predicate Logic, enriched with symbols which satisfy certain needs. Hence, the terms in PROLOG are **variables**, **constants**, and **functions** on variables and constants (Definition 2.2.1). The constants can be **atoms**, **numbers**, or **sequences of special characters**. The fundamental objects of the PROLOG language are the predicates and the lists. We will proceed by examining the terms and the basic objects of PROLOG.

3.3.2 *The Alphabet of PROLOG*

The PROLOG **alphabet** consists of:

- (i) Upper and lower case letters of the English alphabet

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>	<i>G</i>	<i>H</i>	<i>I</i>	<i>J</i>	<i>K</i>	<i>L</i>	<i>M</i>	<i>N</i>	<i>O</i>	<i>P</i>	<i>Q</i>	<i>R</i>	<i>S</i>	<i>T</i>	<i>U</i>	<i>V</i>	<i>W</i>	<i>X</i>	<i>Y</i>	<i>Z</i>
<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>	<i>i</i>	<i>j</i>	<i>k</i>	<i>l</i>	<i>m</i>	<i>n</i>	<i>o</i>	<i>p</i>	<i>q</i>	<i>r</i>	<i>s</i>	<i>t</i>	<i>u</i>	<i>v</i>	<i>w</i>	<i>x</i>	<i>y</i>	<i>z</i>

- (ii) The digits 0 1 2 3 4 5 6 7 8 9

- (iii) Symbols like ! # \$ % ^ & * () _ = - + < > ? / .
depending on the dialect of the PROLOG which we are using.

- (iv) Characters which operate without being directly displayed on the screen, like the space or the new line.

3.3.3 The Variables

The **variables** in PROLOG are letters or words written in capital letters of the PROLOG alphabet. Variables always start with a capital letter or an underscore. For example

`X, Y, Result, Father, Who, _mary, A3`

are variables. On the contrary

`x, y, result, mary`

are NOT variables. Moreover the following are not considered as variables:

<code>1st_athlete</code>	(begins with a digit)
<code>"Result"</code>	(is enclosed in quotes)
<code>3579</code>	(is written with numbers)

Hence, if we write

- (1) `woman(Mother).`
- (2) `woman(mother).`

fact (1) tells PROLOG that every constant which exists in the program is a woman (why?), whereas fact (2) denotes that the specific constant "mother" is a woman.

PROLOG offers one additional possibility, the use of **anonymous**, nameless variables, which are variables whose name we do not need; we simply declare their position with an underscore. For example, writing

`mother(helen,_)`

in PROLOG, we declare that Helen is the mother of some person who is not really of any use for the rest of the program. Asking

`? parents(nick,_,Mother).`

we are only interested in the mother of Nick and we denote his father with an anonymous variable. Most of the time we use an anonymous variable when this specific variable appears in the program only once [Xant90]. As we will see in the relevant section, by using anonymous variables we simplify the process of unification and we save time.

3.3.4 The Constants

The **constants**, which are strings of PROLOG symbols, are divided into three classes, according to their form [Brat90]. Therefore, the constants in PROLOG can be

- (i) *Atomic terms*
- (ii) *Numbers*
- (iii) *Sequences of special characters*

An **atomic term** is a sequence of alphanumerical characters and “_”, hence it consists of letters, digits and probably underscores (but not as the first character because an underscore at the beginning of the sequence denotes a variable!). The starting symbol of an atomic term is always a lower case character. For example,

y342, tom, red_apple

are atoms. The following are NOT atomic terms

Peter	(starts with a capital)
_x5	(starts with an underscore)
3t	(starts with a digit)
a.32	(contains a period)

Any sequence of symbols enclosed in single quotes is also an atom, e.g.:

'x', 'Peter', '5tx', 'John with the nice tie'

The **numbers** are integers or reals, for example

211345, -32, 0.000013, -5.7

Initially, PROLOG was designed to be a language for logic calculations, and its ability to carry out arithmetic operations is relatively limited. However, the most recent edition, PROLOG III [Colm90], has great abilities for mathematical calculations.

Using the **special character sequences** in PROLOG, we construct symbols which have a specific meaning attached. For example,

- - - > or = = = >

can be used as symbols of implication, \rightarrow .

3.3.5 The Predicates

The predicates, Definition 2.2.1, express relations and properties of terms. For example, the predicate

$$\text{greater}(+(X, 2), : (Y, 3))$$

signifies that the term $+(X, 2)$, or equivalently $X + 2$, is greater than the term $:(Y, 3)$ or $Y : 3$, Definition 2.2.1. The symbols “+” and “:” correspond to the addition and division symbols respectively, X and Y are variables, and 2 and 3 are constants. Predicates in PROLOG start with a lower case letter.

In PROLOG, we have the ability to construct **compound predicates**, or **structures**, from predicates which already exist in the program. For example, let us take the facts which describe a family:

```
father(nick).
mother(mary).
child(anne).
child(tim).
```

(*)

In this example, “father”, “mother” and “child” are the predicates, and “nick”, “mary”, “anne” and “tim” are the constants.

The facts (*) can be expressed using only one predicate:

```
family(father(nick), mother(mary), children(anne, tim)).
```

(**)

In the expression (**), “family” is the predicate, but “father”, “mother” and “children” are no longer predicates, but functions [Thay88]. This means that “father(nick)”, “mother(mary)”, and “children(anne, tim)” in (**), as specific values of functions, are considered as constants. Therefore, “father(nick)” in the facts (*) is a different syntactic object from “father(nick)” in (**).

In the same fashion, the fact

```
likes(george, ice_cream).
```

where “likes” is the predicate and “george” and “ice_cream” are constants, can also be expressed using the predicates

```
ice_cream(likes(george))
```

(1)

or

`george(likes(ice_cream))` (2)

where in (1) “ice_cream”, “likes” and “george” are predicate, function and constant respectively, whereas in (2) “george”, “likes” and “ice_cream” are predicate, function and constant respectively.

PROLOG identifies if a given expression is a constant, a function or a predicate, according to the position of the expression in a fact, a rule or a query. This identification method will be seen better in the next subsection, where we will represent predicates as trees.

The great advantage of the use of compound predicates can also be seen in the family example of (*) and (**). Compound predicates offer an abbreviated expression and slick manipulation of the data. For example, if we are interested in the parents of the family, the corresponding queries for program (*) will be:

```
? father(X).
? mother(Y).
```

whereas by using the complex predicate in (**) we would have only one query:

```
? family(father(X),mother(Y), _ ).
```

where the anonymous variable (subsection 3.3.3) indicates that we are not interested in the values of the function “children”.

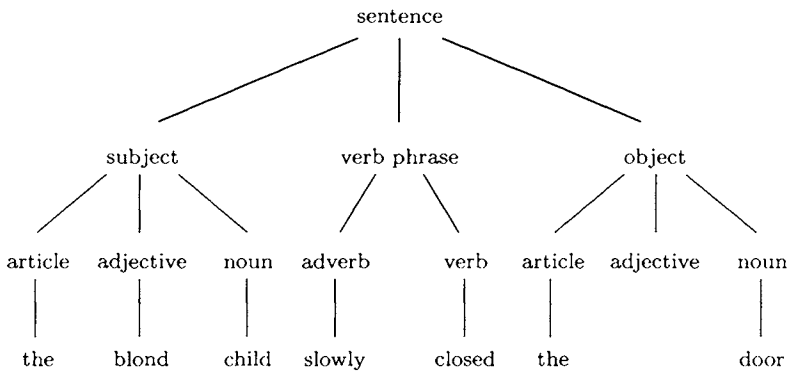
3.3.6 Tree Structure of Predicates

Using a first, simplified approach of syntactic analysis, the sentences in a natural language are composed of a subject, a verb phrase and the object or the predicate. The subject and the object are name expressions which consist of an article, an adjective and a noun, whereas verb phrases consist of a verb and an adverb. Hence the phrase:

the blond child slowly closed the door (1)

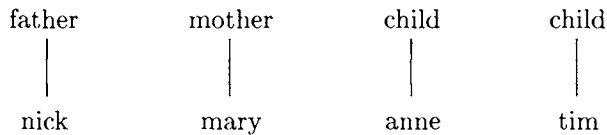
can be analyzed syntactically by the use of a tree [Thay88, Xant90], whose final

nodes (Definition 2.8.9) are the exact words of phrase (1):

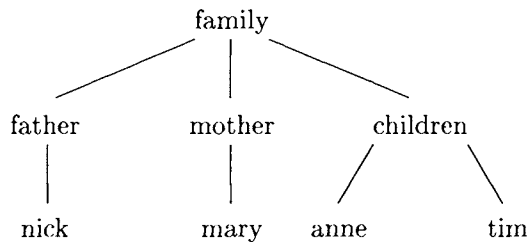


The same technique of syntax analysis is used for the predicates: each predicate corresponds to a tree. The origin of the tree, Definition 2.8.9, is the predicate under consideration. The internal nodes are the functions which are included in the predicate, and the final nodes are the variables and the constants which appear in the functions and the predicate.

Therefore data (*) of the previous section correspond to four trees:

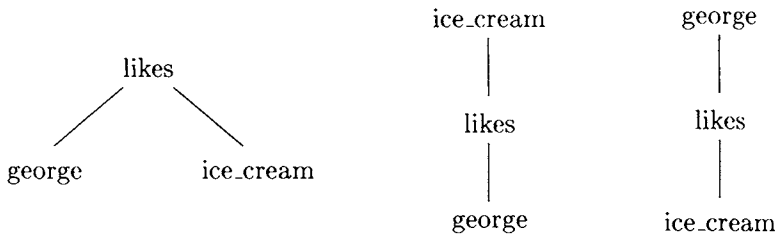


whereas the complex predicate (**) corresponds to the tree:



The expressions “likes(george,ice_cream)”, “ice_cream(likes(george))”,

and “`george(likes(ice_cream))`” have corresponding structures:



By using the tree structure of predicates, PROLOG controls the success of the unification, as we will see in section 3.4. After the unification of two predicates, the corresponding trees must be the same, node by node and edge by edge.

3.3.7 The Lists

Lists are simple catalogues of data enclosed in “`[`” and “`]`”, for example:

```
[wine, soap, john, X, house].
```

These lists, which constitute a fundamental building block of LISP, [WiBe89], have been embodied in PROLOG. Using lists we can collect a lot of information in one PROLOG object. For example, the facts:

```
likes(george, football).
likes(george, food).
likes(george, wine).
```

can be given using a single list as the fact:

```
likes(george, [football, food, wine]).
```

This means that the sequence of terms, `football`, `food`, `wine`, is represented by the object:

```
[football, food, wine]
```

In general, a list is composed of a sequence of terms.

By using lists we achieve space efficiency and greater slickness when managing complex data.

Lists are generally defined recursively:

- (i) The **empty list**, i.e., a list which does not contain any elements, is represented by “[]”.
- (ii) A **non-empty list** is composed of two terms:
 - (1) The first term is called the **head** of the list.
 - (2) The second term consists of the rest of the list, and is called the **tail** or **body** of the list

The head of a list can be an arbitrary PROLOG term, a constant, a variable or a function. *The tail of a list has to be a list.* For example, the head of the list:

[football, food, wine]

is the constant, `football`; and its tail is the list:

[food, wine]

Due to the recursive definition of functions in Predicate Logic, Definition 2.2.1, lists are not functions. Moreover, list elements are always selected based on a relation or property they possess. For example, the list:

[meat, bread]

may denote objects which have to be bought, in which case it is expressed by the predicate:

shopping(meat, bread)

Therefore a list can be considered as a predicate by the use of a **predicator**, “.”, i.e., a special symbol denoting the existence of some predicate at the corresponding place. For example, by writing

.(salad, vinegar)

we denote that we made up the list:

[salad, vinegar]

without specifying either the qualities of the terms “salad” and “vinegar”, or the relationship between them. Hence “.(salad,vinegar)” is considered to be a **generalized predicate**, a predicate whose form we do not know precisely. Therefore, we cannot query PROLOG:

```

        ? [salad,vinegar].
or      ? [X,Y].
or      ? [X,vinegar].

```

because we do not know the predicate which corresponds to “.”. Furthermore, within the same program, more than one list can exist, all of them being expressed by the special symbol “.”.

In general, a list can be written as a predicate:

```
.(Head,Tail)
```

The tail is also a list with a head and tail. If we also use “.” as a **functor**, i.e., a symbol denoting the existence of some function at the corresponding place, then the general form of a list is:

```
.(Head1,.(Head2,.(Head3,...(Head,[ ])...))
```

where [] denotes the empty list.

For example the list [a,b,c] has the general form:

```
.(a,.(b,.(c,[ ])))
```

A list containing only one element has the form:

```

[element]
or      .(element,[ ])

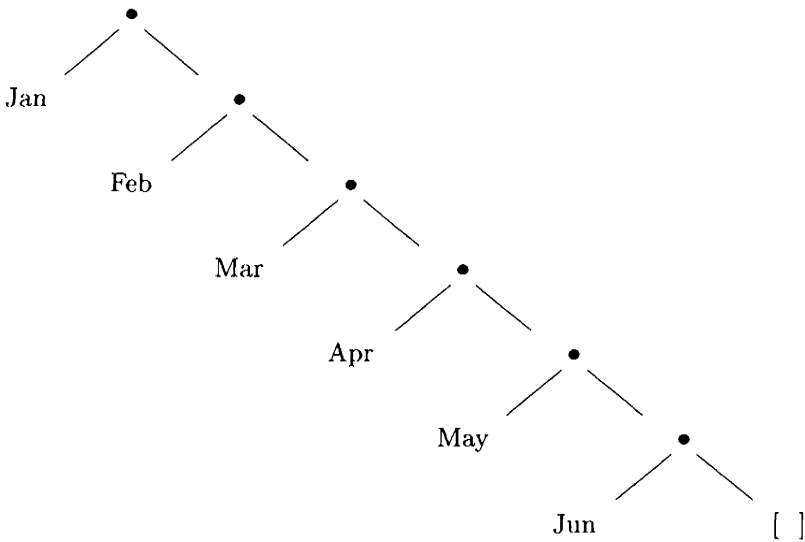
```

For the typical construction of a list we use trees. The tree structure of a list forms the **internal** representation of lists with respect to the PROLOG interpreter.

For example, the tree structure of the list:

[Jan, Feb, Mar, Apr, May, Jun]

is given by:



where “.” as the origin of the tree is the special symbol for a predicate, and “.” in the internal nodes is a symbol of a function.

All PROLOG editions offer a slick notation for lists which facilitates the representation of long or variable length lists. This representation is based on the use of symbols “ [”, “] ” and “ : ”. For example, the list:

[*a*, *b*, *c*, *d*]

with head *a* and tail the list [*b*, *c*, *d*], is written as:

[*H* : *T*]

where the variables *H* and *T* correspond to the terms *a* and [*b*, *c*, *d*] respectively.

3.4 Operation Mechanism

3.4.1 *The Unification Procedure in PROLOG*

The **unification** procedure of terms, section 2.9, with its algorithmic nature, subsection 2.9.9, is one of the most important operations in PROLOG.

The unification procedure does not exist in other high level languages. Although LISP uses unification as its fundamental operation property, it does not use the resolution method as a proof mechanism. However, the inference power of PROLOG comes from the smooth cooperation of unification procedures and resolution.

Unification enables the update of variables with given PROLOG objects. For example, the reply of PROLOG

`X = football`

means that the variable *X* has taken a value, has been **updated** by the constant “football”.

In general, with unification we examine if two predicates can be identical. If they cannot be identical, the unification procedure fails. If they can be identical, unification succeeds and the result of the unification is the update of the variables of the two predicates with such values that the two predicates match.

In the bibliography, the word **matching** is often used for **unification** and the words **binding** and **instantiation** for the word **update**. Binding in this case is not the binding of free variables as in Definitions 2.2.12 and 2.2.13, because in any case, all variables in PROLOG are bound with universal quantifiers, Definitions 2.4.1 and 2.4.4. On the contrary, the term refers to the fact that the variables take a certain value through the procedures of the program.

By means of the unification algorithm and the general unifier, Algorithm 2.9.7 and Definition 2.9.4, PROLOG can execute complex substitutions. This procedure can be marvelously represented using the tree structure of predicates:

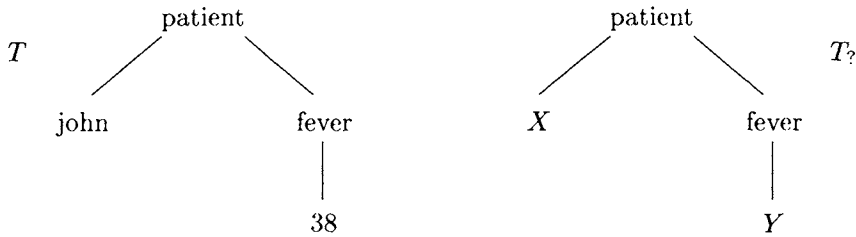
Let us suppose that a database in PROLOG contains the fact:

`patient(john, fever(38)).` (*)

and that the following query is made:

`? patient(X, fever(Y)).`

The tree structures, T and $T_?$, of fact $(*)$ and the query respectively, are:



If we bring tree $T_?$ directly over tree T , in order to achieve exact matching, then the variables of tree $T_?$ have to be unified, to match, with the terms to which they correspond, since the predicate and the number of edges are the same in both trees. Thus the reply of PROLOG in the above query is:

$X = \text{john}$

$Y = 38$

In general, the rules which are obeyed during the procedure of unification to check if A and B are identical [Xant90] are:

- (1) If A and B are constants, then A and B match only if they correspond to the same object.
- (2) If A is a variable and B is anything, then A matches B . If B is a variable, then it matches A .
- (3) If A and B are functions, then A and B match only if:
 - (a) A and B have the same initial function, and
 - (b) the rest of the corresponding subterms of A and B match.
- (4) If A and B are predicates, then A and B match if and only if:
 - (a) A and B have the same initial predicate, and
 - (b) the rest of the corresponding terms of A and B match.

The above matching rules result in the **recursive nature** of the unification procedure. Rules (3) and (4), more specifically parts (3b) and (4b), refer to steps 1, 2, 3 and 4 in the informal description of unification in section 2.9. The recursive nature of the above rules can also be observed in the following example.

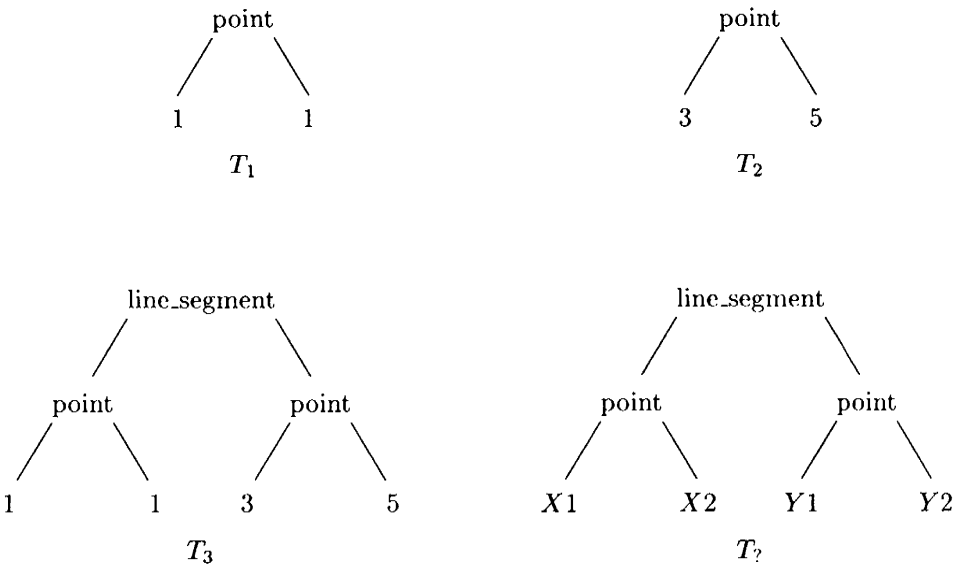
We are given the following geometric data:

```
point(1,1).
point(3,5).
line_segment(point(1,1),point(3,5)).
```

and the query:

```
? line_segment(point(X1,X2),point(Y1,Y2)).
```

Their corresponding tree structures are:



The unification procedure will follow the rule:

- (i) line_segment matches line_segment (4a)
- (ii) point matches point (4b, 3a in (i))
- (iii) X1 matches 1 (3b in (ii))
- (iv) point matches point (4b, 3a in (i))
- (v) Y1 matches 3 (3b in (ii))
- Y2 matches 5 (3b in (ii))

Therefore trees T_3 and T_7 are unified, or match, for the following values of the variables:

$$X1 = 1$$

$$X2 = 1$$

$$Y1 = 3$$

$$Y2 = 5$$

The recursive definitions of relations constitute one of the dynamic characteristics of PROLOG and will be presented in detail in subsection 3.4.5.

3.4.2 Inference and the Backtracking Procedure

As we have already mentioned, a program in PROLOG has two interpretations: the **logic** or **declarative** one, and the **procedural** one. In this subsection we will examine the procedural interpretation of PROLOG. We will describe *how* PROLOG derives conclusions.

We have already presented the unification procedure and the resolution method. Applying the unification process, we end up with a set of ground terms of the program, Definition 2.2.15. In the next step, we repeatedly apply resolution to this set of ground terms, and thus reach conclusions by proving the empty clause. By using the resolution method, we are guaranteed that any proposition which can take a truth assignment, will take it. However, in PROLOG, we cannot direct the inference in sentences in which we are interested, and this is of great importance for the efficiency of the programs.

For example, let us consider the following propositions in Propositional Logic:

$$(1) \quad D$$

$$(2) \quad E$$

$$(3) \quad B \vee \neg E$$

$$(4) \quad A \vee \neg D$$

$$(5) \quad A \vee \neg B$$

and let us assume that we posed the problem when A holds. We introduce the proposition:

$$(6) \quad \neg A$$

and we have the following proof through resolution of A :

(7)	B	from (2) and (3)	
(8)	A	from (7) and (5)	(*)
(9)	\square	from (8) and (6)	

However, A can also be proved by the following resolution proof:

(7')	A	from (1) and (4)	
(8')	\square	from (7') and (6)	(**)

Proof (*) consists of more steps than (**), and its implementation takes more time. Furthermore, in proof (*) we had to conclude proposition B , which was not related to A .

PROLOG embodies in its inference mechanism a special search procedure in the universe of terms of a program. On the one hand, this procedure gives the ability to the programmer to control the sequence of calculations; and on the other hand it guarantees that all possible solutions of a problem will be found [Lloy87]. This procedure is called **backtracking**. For example, consider the following PROLOG database:

```
thief(john).                /* 1 */
likes(mary,food).           /* 2 */
likes(mary,wine).           /* 3 */
likes(john,X):- likes(X,wine). /* 4 */
can_steal(X,Y):- thief(X), likes(X,Y). /* 5 */
```

(In most PROLOG editions we use the character sequences “/*” to introduce and “*/” to end comments, non-executable by the program. Therefore in the above program we have introduced the numbering of clauses as comments).

Let us suppose that we introduce the goal query:

```
can_steal(john,Z).          /* what can john steal */
```

The steps followed to satisfy the goal are:

Step 1.

? can_steal(john, Z).

X/john

Z/Y

thief(john)

(5)

The database is examined in order to unify the goal with a fact or with the head of a rule. Unification succeeds with the head of rule (5) and with the substitutions X/john , Z/Y . The position of rule (5) in the database is recorded. The variables appearing in the clauses of the head of rule (5) are also updated through the same substitutions.

Step 2.

? thief(john).

yes

(1)

PROLOG aims to satisfy the subgoals, one at a time.

The database is examined from the top, clause (1), for the unification of the subgoal. Unification succeeds with clause (1). This clause does not contain any variables, so the subgoal is satisfied.

Step 3.

? likes(john, Y).

X/Y

likes(Y, wine)

(4)

PROLOG aims to satisfy the second subgoal. The database is examined from the top for the unification of the subgoal. Unification succeeds with rule (4) and through the substitution X/Y .

The variables of the predicate in the body of rule (4) are immediately updated.

Step 4.

? likes(Y, wine).

The clause of the body of rule (4) is acted upon to be satisfied. The database is searched once more beginning from the top in order to succeed the unification of the goal.

Step 5.

Y/mary

The unification attempt with (2) fails because "food" and "wine" cannot be unified. The position of the clause with which it was attempted to unify, (2), is recorded.

(2)

Y/mary

Step 6.

PROLOG attempts to satisfy the goal in a different manner and goes on to the next clause of the program. Unification succeeds with (3).

(3)

no

yes

Step 7:

The initial goal, “`can_steal(john,Z)`”, has been satisfied by means of the substitution Z/mary . PROLOG now attempts to satisfy the goal in a different way. For that purpose, *it backtracks to the immediately previous subgoal* that it has satisfied, “`likes(Y,wine)`”, in order to unify it with a different Horn clause from the program. At this point, PROLOG *frees all the variables from their updated values*. Then, it checks the clauses in the database which are *deeper* than the last clause with which it unified the subgoal, step 6, clause (3). Deeper in PROLOG means that the corresponding clause has a larger serial number than the last clause with which unification was performed, subsection 3.1.2. Since the position in the database of the clause being executed is recorded in every computational step, PROLOG has the ability to spot the deeper clauses. Since there is no fact or rule, the head of which is unified with the subgoal, PROLOG finishes its search.

In the above description of program execution, we can see that PROLOG aims to satisfy *one goal at every computational step*. This property of the derivation mechanism of PROLOG, and more specifically of the backtracking procedure, gives PROLOG a **deterministic** nature.

The order in which the programming clauses have been entered in the database plays an important role in the execution and control of a PROLOG program. The attempt of PROLOG to satisfy the (sub)goals of the program by searching in depth the database, implements one of the basic algorithmic tree searching strategies, the **depth-first search strategy**. This algorithm, which constitutes the heart of the backtracking procedure, will now be presented.

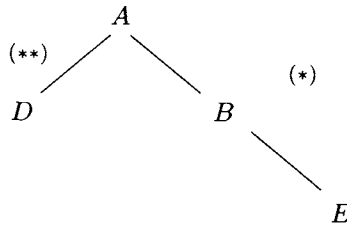
3.4.3 Depth-first Search

The use of trees in finding the solutions to a problem is based on the representation of the initial data, as well as the solutions, by a **state space** [Nils71]. A state space is a tree in which:

- (a) The nodes correspond to the different states of a problem: the **initial state**, the **intermediate states** during its resolution process and the **final states**.
- (b) The edges between the nodes correspond to the **allowed transitions** from one state to another.

Hence, the solution to a problem is reduced to *finding a path from the initial state to the final state* [Brat90].

During the procedure of satisfying a possibly complex goal, PROLOG searches a state space. The goal is the origin of the corresponding tree. The intermediate nodes correspond to the intermediate subgoals which are to be achieved each time. The final states correspond to facts and/or consequences (Definition 2.5.21) of the facts in the database. Choosing the way of getting from one state to another is equivalent to choosing the corresponding subgoals. Therefore, the state space of the program in the first example of subsection 3.2.2 is described with the following tree:



where clause A is the initial state, the goal of the problem, and clauses D and E are the final states, the facts of the database. Therefore, for the solution of the problem, which is conclusion A , there are two different solution paths with, as corresponding sets of allowed transitions, the sets:

$$\begin{array}{ll}
 (**) & \{D \longrightarrow A\} \\
 \text{and} & (*) \quad \{E \longrightarrow B, B \longrightarrow A\}
 \end{array}$$

Given the state space of a problem, the question is how to select the appropriate **solution path** of the problem and, furthermore, how to find **all** the alternative solution paths. The answer to these questions can be given using the **depth-first search algorithm** for tree traversal.

The main idea for the construction of the depth-first search algorithm is based on the following observations.

In order to find a solution path, `Solution_Path`, from a node N of a tree to a **final** state node:

- (a) If N is a final state node, then `Solution_Path` is the list $[N]$.
- (b) If there exists a node $N1$, a descendant of N , such that there exists a solution path, `Solution_Path_1`, from $N1$ to the solution node, then `Solution_Path` is the list $[N : \text{Solution_Path}_1]$.

This algorithm is executed repeatedly, until all the possible solution paths are found. Every descendant node which is examined to see whether it belongs to a solution path or not, must be *deeper* and *more to the left* in the tree, compared to the initial node, than its predecessor; hence the name of the algorithm. When PROLOG needs to backtrack in order to find alternative paths, it **backtracks** to the immediately preceding node.

Consequently, the depth-first search algorithm is given in the following PROLOG program:

```
find_path(N,[N]): goal(N).
find_path(N,[N:Solution_Path]):- allowed_transition(N,N1),
                                find_path(N1,Solution_Path1).
```

In this program, and more specifically in the second clause, it may seem strange that we use the predicate “`find_path`” inside the definition of “`find_path`”! Such **recursive definitions of relations** between objects are used extensively in PROLOG, and will be presented in detail in subsection 3.4.5.

The ability to define in PROLOG its own structural and functional elements, such as the depth-first search algorithm and recursion, constitutes one of the language’s dynamic characteristics. This ability is based on the declarative and logic nature of PROLOG programs and is called PROLOG *through* PROLOG!

3.4.4 Controlling Backtracking: *Cut*

The PROLOG language offers a special predicate, the **cut**, symbolized by “!”, which controls the inference procedure and, more specifically, backtracking. The “cut” is used to prevent PROLOG from taking certain paths while searching the state space of a problem. This is done either in cases where the user knows that they are not solution paths of the given problem, or in order to make computation time shorter, or finally because it is not considered imperative to deduce conclusions for some specific facts (a form of rejecting facts).

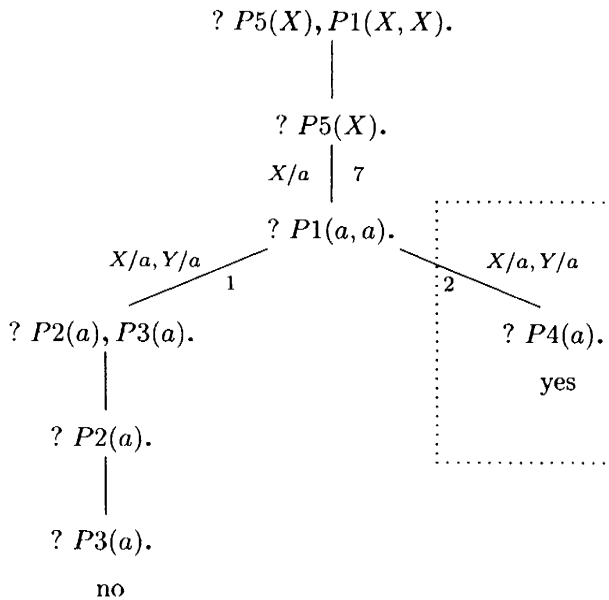
As a special predicate, “cut” is always true. That means that when it is set to be satisfied, it has to be satisfied. It is interesting to examine the consequences of using “cut” on the execution and the set of conclusions of a PROLOG program. Let us consider for example the following PROLOG database [Thay88]:

$P1(X, Y) :- P2(X), P3(Y).$	$/ * 1 * /$
$P2(X, Y) :- P4(Y).$	$/ * 2 * /$
$P2(a).$	$/ * 3 * /$
$P3(b).$	$/ * 4 * /$
$P4(a).$	$/ * 5 * /$
$P5(a).$	$/ * 6 * /$
$P5(b).$	$/ * 7 * /$

and the query:

$? P5(X), P1(X, X).$

The state space, and the only solution path of the problem, are described by the tree:



PROLOG satisfies the initial goal “ $P5(X), P1(X, X).$ ” using the substitution X/a .

Let us see now what will happen if we use “cut” in the body of programming clause $/ * 1 * /$, so that:

$P1(X, Y) :- P2(X), !, P3(Y).$	$/ * 1' * /$
--------------------------------	--------------

PROLOG will attempt to satisfy the initial goal following once more the left path. Hence, it will try to satisfy subgoal “ $P1(a, a)$ ” by activating clause $/ * 1 * /$. After it satisfies $P2(a)$, it will come across the cut, $!$, which it will satisfy immediately, because “cut” is always true. It will therefore go ahead and try to unify $P3(a)$ and will fail, because there is no Horn clause in the program to be unified with $P3(a)$. Normally it should now backtrack to subgoal “ $P1(a, a)$ ”, for the purpose of satisfying it through the activation of clause $/ * 2 * /$. Since, however, it found the cut in the body of clause $/ * 1' * /$, it will not backtrack. *! prevents PROLOG from re-rendering as a subgoal the head of a clause in whose body there is a “cut”.* Hence, although in the case of clause $/ * 1 * /$ the initial goal was satisfied through the right solution path with X/a and Y/a , in the case of clause $/ * 1' * /$ it is not satisfied, which means that PROLOG will reply “no”. In other words, the right solution path is cut off from the state space (dotted line). In more complex state spaces, the use of “cut” can result in cutting off entire subtrees of the space. The “cut” predicate was named after this functional nature of cutting off subtrees.

It is interesting how, using “cut” in a PROLOG program, we can get rid of loops, which are never ending operations, which might be caused by ill-defined data. Let us examine the following PROLOG program:

```
thief(john).                / * 1 * /
thief(X):- thief(X).        / * 2 * /
```

and the query

```
? thief(X).                / * 3 * /
```

Then by unification from clause $/ * 1 * /$, PROLOG immediately finds:

$X = \text{john}$

and after freeing variable X from the value “john”, it attempts to satisfy the goal in a different way, by means of $/ * 2 * /$. After the resolution, the new goal is

```
? thief(X).                / * 4 * /
```

which matches the initial goal $/ * 3 * /$. Clause $/ * 4 * /$ is unified with the head of rule $/ * 2 * /$, and after the unification, the new goal is again $/ * 4 * /$, which means that the program is in an endless loop, from which it cannot escape.

Let us study now the same PROLOG program, but with a cut in clause / * 2 * /:

```
thief(john).                / * 1 * /
thief(X):- thief(X),!.      / * 2' * /
```

and the query:

```
? thief(X).                / * 3 * /
```

Clause / * 1 * / immediately gives the value:

$X = \text{john}$

PROLOG tries to find all the possible solutions; unifies with the head of clause / * 2 * /, solves, finds a new goal:

```
? thief(X).                / * 4 * /
```

and stops, because “cut” does not permit the setting for a second time as goal the head of rule / * 2' * /, in whose body the cut is. Hence, the only answer given is

$X = \text{john}$

and the program is carried out without an endless loop.

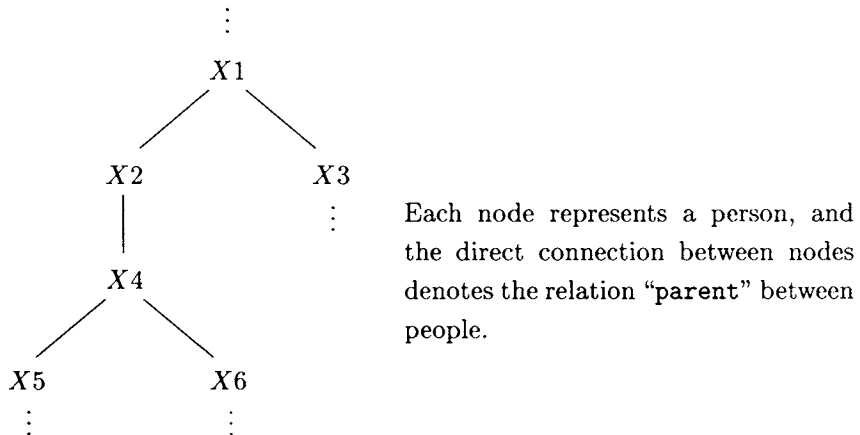
The function of “cut” can be described purely in terms of the procedural interpretation of a PROLOG program, meaning that its use and the control of its execution is left to the programmer. The ability to use “cut” as an element of the PROLOG language is currently under dispute, mostly by those who want PROLOG to be a purely logic programming language. This dispute is based mainly on the following reasons:

- (1) A PROLOG program without “cut” is able to find through backtracking all the correct answers to a given query (completeness of computation of correct answers). The presence of “cut” destroys this form of completeness, since “cut” does not allow the program to calculate all the possible correct replies [Lloy87].
- (2) The use of “cut” allows us to write a logically incorrect PROLOG program, which, although it allows the derivation of incorrect conclusions, behaves correctly using “cut”.

However, the use of “cut” in certain programs can improve their efficiency without diminishing their logical clarity.

3.4.5 Recursive Definitions in PROLOG

Using **recursive definitions** we can define new predicates in a PROLOG program. For example, let us assume that we want to define the relation “ancestor” in order to examine the following part of a family tree:



We observe that:

for all X, Z in the family tree, when X and Z are directly connected,
 X is an ancestor of Z if X is a parent of Z .

We can define the ancestor relation using a Horn clause:

$\text{ancestor}(X, Z) \text{ :- parent}(X, Z).$ (a)

Therefore, by using the predicate “parent”, we defined the predicate “ancestor” for people who are one generation apart.

For the ancestor relation between people who are two generations apart, for example between $X1$ and $X4$, we observe that:

For all X, Z in the family tree, X is an ancestor of Z if X is a parent
of some Y and Y is a parent of Z .

Hence we write this clause:

$\text{ancestor}(X, Z) \text{ :- parent}(X, Y), \text{ parent}(Y, Z).$ (b)

In our effort to check the “ancestor” relation between people who are more than two generations apart, we will need to write a variable length sequence of programming clauses:

```

ancestor(X,Z):- parent(X,Y1), parent(Y1,Y2), parent(Y2,Z).
ancestor(X,Z):- parent(X,Y1), parent(Y1,Y2),
                  parent(Y2,Y3), parent(Y3,Z).

```

Each time we go deeper into the family tree we have to define a new rule. This is definitely not effective for our programs, because a program in no matter what language, is useful and effective only if it solves general and not just specific instances of a problem.

There is a more efficient way to define the “ancestor” relation:

We observe that:

For all X and Z in the family tree, X is an ancestor of Z if X is the parent of some Y and Y is an ancestor of Z

Thus, we can write the following PROLOG programming clause:

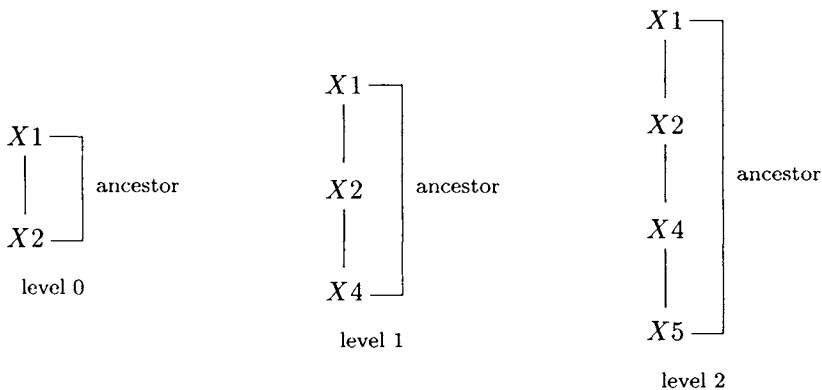
```

ancestor(X,Z):- parent(X,Y), ancestor(Y,Z).

```

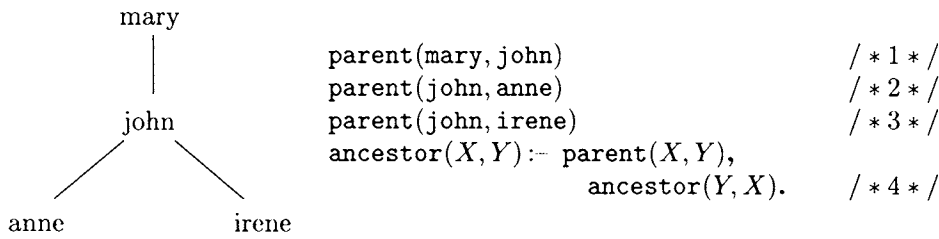
(c)

In this definition, we used the predicate “ancestor” to define the predicate “ancestor”! Definitions of this form are called **recursive definitions**, and the relations they denote are called **recursive relations**. Intuitively, we would say that *recursion is born the moment when a sketch is built from the reproduction of itself in a different scale or a different level* [Xant90]. Hence, in the family tree example, the control of the “ancestor” relation between $X1$ and $X5$ is reproduced in the levels of the following diagram:



Rule (c) does not suffice for the definition of the “ancestor” relation. Although it logically expresses whatever we want to define, it does not give the program the information necessary for the answer to queries on the relation “ancestor”.

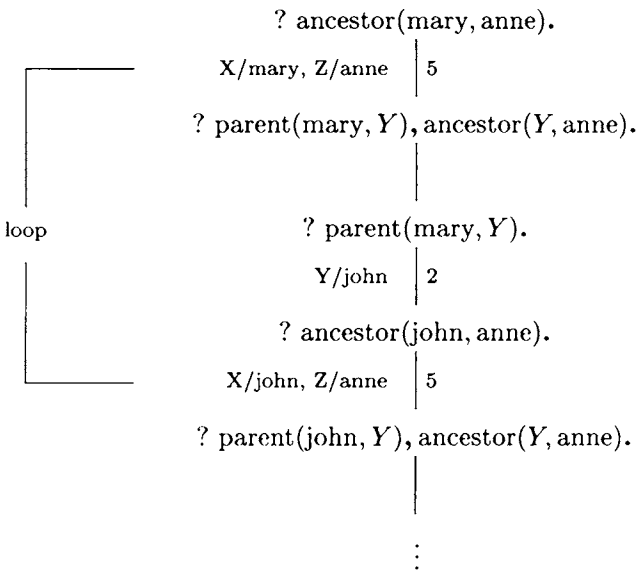
Let us see an example. We are given the following family tree and the corresponding PROLOG database:



Assume that the following query is asked:

? ancestor(mary, anne).

PROLOG will attempt to answer this query, searching the state space depicted by the following tree:



PROLOG fails to finish the search in the problem's state space. The tree's branchings will continue for ever and PROLOG will always be inside an endless loop trying to satisfy the goal "ancestor(X, Z)", for different updates of the variables each time. Of course PROLOG will not stop the execution of the program *without giving a "yes" or "no" answer*. The halting of the program is caused by the limitation of the machine on which this specific edition of PROLOG is running and not by the natural termination of a PROLOG program.

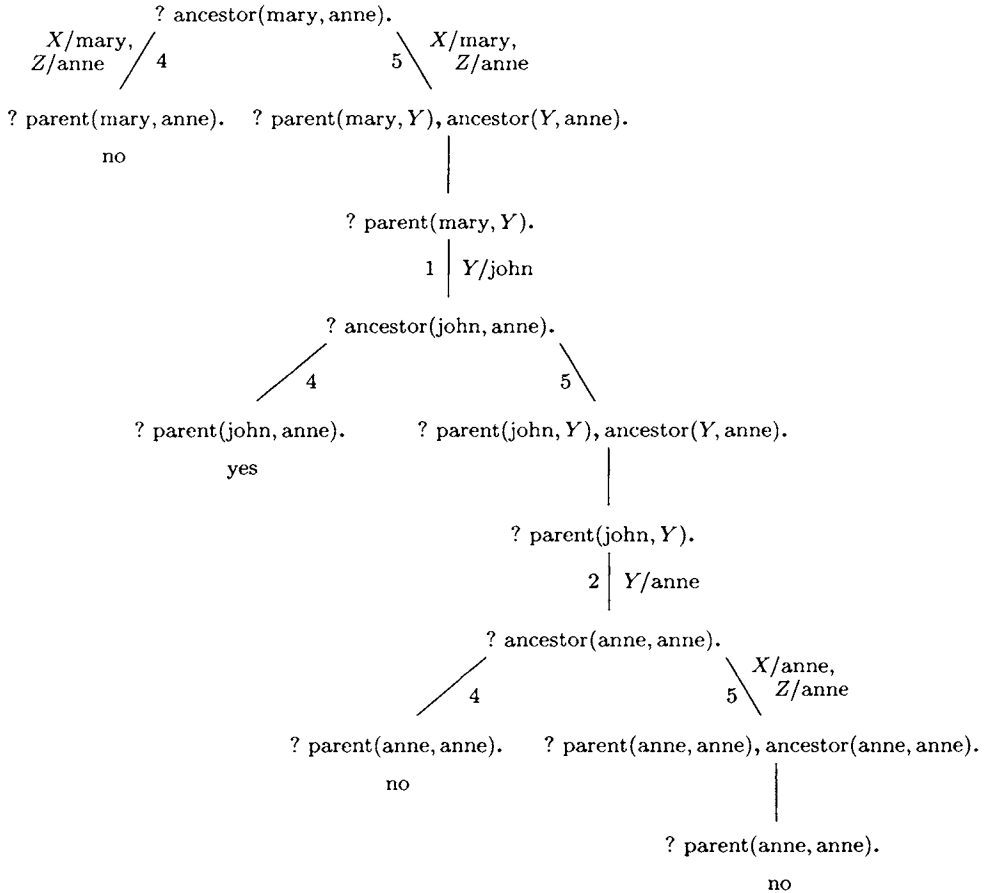
We therefore need a method which *guarantees* the termination of the execution of a given recursive procedure. Such a method can be implemented by using a different Horn clause which states the **marginal conditions** under which a relation holds. In the example of the family tree, this condition is the relation “parent”, meaning that an ancestor relation between two persons holds marginally when one is the parent of the other. This means that:

For all X, Z in the family tree, X is an ancestor of Z if X is a parent of Z .

We need therefore to add in the database the programming clause:

$\text{ancestor}(X, Z) \text{ :- } \text{parent}(X, Z).$ /* 4 */

Clause /* 4 */ acts also as a **boundary condition** of the defined relation. With the introduction of /* 4 */, the corresponding state space of the reply in the previous query is depicted by the tree:



Thus, by introducing clause $/ * 4 * /$, PROLOG ends the search of the state space and replies positively to the query.

The marginal condition of a recursive relation is also called the **bound of the recursion**. The choice of the bound of a recursive relation and the corresponding programming clause has to be made very carefully. In general, the principle followed for problems which can have a recursive solution is:

We divide the problem into two subgroups:

- (a) The first subgroup contains the “simple” instance of the problem (the bound of the recursion).
- (b) The second subgroup contains the “general” problem instances whose solutions are found by reducing them to simple versions of the problem (recursion).

Before getting into the presentation of specific management operations of lists, it is worthwhile to mention the unification operation of lists.

3.4.6 List Management

Since lists are abstract predicates, subsection 3.2.4, they are unified just like the predicates. In practice:

A list $L1$ is unifiable with a list $L2$ if $L1$ and $L2$ have the same number of elements, and all the internal elements of $L1$ are unifiable with the corresponding internal elements of $L2$.

Therefore two lists can be unified if and only if their heads and their tails, i.e., the elements of the lists, can be unified. For example, for the lists:

$$L1 = [a : [b, c]]$$

and

$$L2 = [a : Y]$$

the PROLOG query:

$$? L1 = L2.$$

succeeds with the substitution $Y = [b, c]$. On the contrary, lists:

$$L1 = [a, b]$$

and

$$L2 = [\text{height}, \text{color}, \text{position}]$$

do not unify because they do not have the same number of elements.

Lists can be used in the representation of sets. However, there is one basic difference. The order of the elements in a set is not important, whereas the order of the elements in a list is crucial. The principal operations between sets and lists are similar. Therefore, based on the recursive definition of lists and the unification procedure, we can define operations such as:

- (1) Examine if an element or a list of elements belong to a list, i.e., the subset relation for lists.
- (2) Concatenation of lists, to define the union operation for lists.

3.4.6.1 The member predicate

The **member** predicate is used to examine if an element is part of a list. This predicate is a built-in predicate in most PROLOG editions.

In general:

Element X is a member of list L if

- (a) X is the head of L , or
- (b) X is a member of the tail of L .

We can thus recursively define the **member** predicate by:

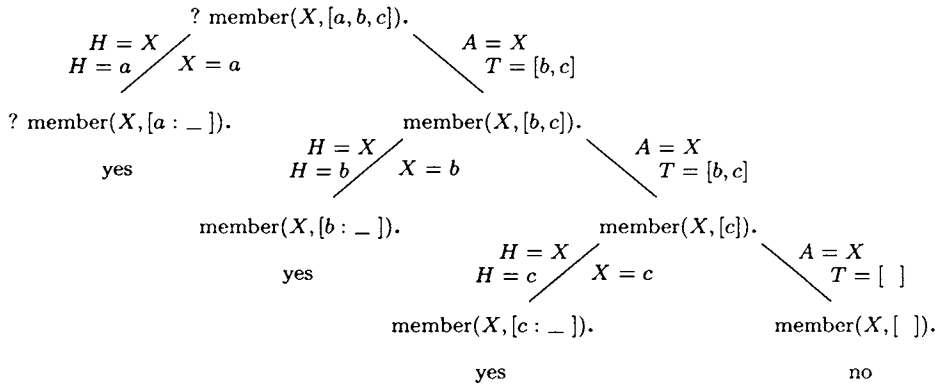
- (1) `member(H , [H : _]).`
- (2) `member(A , [_ : T]) :- member(A , T).`

where “_” is the anonymous variable. Clause (1) is then the bound of the recursion.

For example, for the PROLOG query:

`? member(X, [a, b, c]).`

the state space is depicted by the tree:



Therefore, element X is a member of the list $[a, b, c]$ only if

$X = a$ or
 $X = b$ or
 $X = c$

We observe therefore that PROLOG, attempting to satisfy the query:

`? member(X, [a, b, c]).`

instantiates variable X with concrete constants.

3.4.6.2 The append predicate

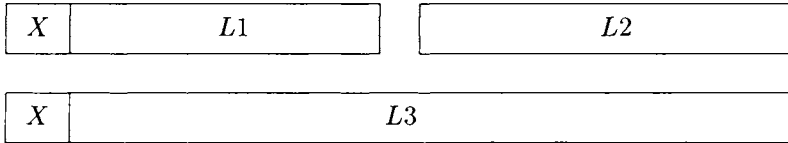
With the use of the `append` predicate we construct a list $L3$ as a result of the concatenation of two lists $L1$ and $L2$. Thus we create a new predicate, `append(L1, L2, L3)`.

To define `append` we observe the following:

- (a) The result of the concatenation of a list with the empty list is the initial list, therefore,

`append([], L, L)` /* 1 */

- (b) The first list is of the form $[X : L1]$. If we concatenate this list with list $L2$, the result will be a list which has the same head as the first list body, and its tail will be the result of the concatenation of the tail of the first list with $L2$. These observations can be better understood with the following diagram:



Based on the above observation we can write

`append([X : L1], L2, [X : L3]) :- append(L1, L2, L3) /* 2 */`

It is clear that clauses `/* 1 */` and `/* 2 */` form the recursive definition of the `append` predicate. `/* 1 */` is the bound of the recursion, and `/* 2 */` is the reduction of the problem of concatenating two complex lists to the concatenation of simpler lists, in other words, the recursion.

Let us look at an example. Suppose we are given the query:

? `append([a, b], [1, 2], [a, b, 1, 2]).`

The state space of the answer to the query is depicted by the following tree:

? `append([a, b], [1, 2], [a, b, 1, 2]).`

$$2 \left| \begin{array}{l} X/a, L1/[b], \\ L2/[1, 2], L3/[b, 1, 2] \end{array} \right.$$

? `append([b], [1, 2], [b, 1, 2]).`

$$2 \left| \begin{array}{l} X/b, L1/[], \\ L2/[1, 2], L3/[1, 2] \end{array} \right.$$

? `append([], [1, 2], [1, 2]).`

yes
from (1) and for $L/[1, 2]$

Hence there is a solution path, so PROLOG will give a positive answer to the query.

The `append` predicate is a built-in predicate in most PROLOG editions. Some other predicates which are used for list manipulation are presented in the exercises.

3.5 Built-in Predicates

Built-in predicates are predicates defined within PROLOG and used in the management of data and the interaction of the user with PROLOG.

3.5.1 Data Management Predicates: **assert** and **retract**

A PROLOG program is a database with the facts and rules of the program as the data. Any change in the data of the program requires the update of the PROLOG program itself. In the traditional programming languages, updating the program, i.e., adding and deleting data, as well as modifying the flow control of the program, is carried out by the programmer. On the contrary, in PROLOG, the program is updated automatically during its execution, using the specific built-in predicates, **assert** and **retract**.

The **assert** predicate is used for the addition of facts and rules in the program. Its syntax is:

assert(*G*)

where *G* is any Horn clause. The **assert**(*G*) predicate *always* succeeds, and has as a result the addition of *G* to the data of the program.

This predicate, **assert**, can be used in different ways. Let us take for example the following program which consists of a single clause:

person(george). /* 1 */

If we ask PROLOG the query:

? **mortal**(george).

we will get the answer “no”, because in the program there is no relation defined for mortals. If now we state the query:

? **assert**(**mortal**(george)).

then, because **assert** always succeeds, the programming clause:

mortal(george). /* 2 */

will be embodied in the program, resulting in the success of the query.

Moreover, `assert` can be used to enter a new rule. In the previous example, instead of stating the query:

```
? assert(mortal(george)).
```

we can state

```
? assert(mortal(X):- person(X)).
```

The result of the above query will be the introduction of the rule:

```
mortal(X):- person(X).           /* 2' */
```

instead of fact `/* 2 */`.

The position of introduction of a clause in the database can be chosen using the following variations of the `assert` predicate:

```
asserta(G)      and      assertz(G)
```

which introduce the clause G at the beginning or at the end of the program data, respectively.

Just as we introduce data to the program with `assert`, we can delete data using the built-in predicate:

```
retract(G)
```

where G is a Horn clause. The operation of `retract` is the opposite of the operation of `assert`.

3.5.2 Interaction Predicates: `read`, `write` and `consult`

PROLOG, as a slick programming language, offers the ability of interaction of a program with the peripheral units of the computer; the screen and the user. This is done through a number of specific built-in predicates, like `read`, `write` and `consult`.

PROLOG will wait until we enter two character strings, which it will unify with the variables `Name` and `Serial_number`, respectively. Then, it will print both of these character strings on the same line, separated by the two spaces due to `write(' ' '')`. After that, it will execute the specialized predicate `nl`, which puts the cursor at the beginning of the next screen or printer line.

With the predicate `consult(file)` we have the ability to use a PROLOG program from a file.

Predicate `consult(file)` always succeeds, resulting in saving the clauses which are in “file” in the computer’s dynamic memory, so that they can be used from PROLOG to derive conclusions.

3.5.3 Equality in PROLOG: Predicates `=`, `==`, `:=`, `is`, `=\=`, `\==`

PROLOG uses different equality predicates to examine the equality of objects in its language. The most used equality predicates, which are common in the different editions of PROLOG, are:

“`=`” Where clause $A = B$ succeeds when the objects (constants, variables, atoms or structures) A and B match.

“`==`” Where clause $A == B$ succeeds if the objects A and B are identical in all respects. For example, equality:

`father(paul,andy) = father(paul,X)`

succeeds by means of the substitution $X/andy$, while equality:

`father(paul,andy) == father(paul,X)`

does not succeed.

“`:=`” Where clause “ $A1 := A2$ ” succeeds when $A1$ and $A2$ are numerical expressions which are equal.

“`is`” Where clause “ $X \text{ is } A$ ” succeeds when X is a variable and A is a numerical expression. With predicate “ $X \text{ is } A$ ”, PROLOG finds the value of A and unifies it with variable X . Here we have to stress the difference between “`is`” and “`=`”. For example, the reply of PROLOG

to the query:

$? X = 3 - 1.$

will be $X = 3 - 1$. This means that if we use “=”, the operation in the right hand side of the equation is not executed. If however we ask:

$? X \text{ is } 3 - 1.$

then the reply will be

$X = 2$

“\==” The predicate “\==” is the negation of “==”, which means that clause $A \text{ \textbackslash == } B$ succeeds when $A == B$ fails.

“=\=” The predicate “=\=” is the negation of “:=”, therefore programming clause “ $A1 \text{ \textbackslash = } A2$ ” succeeds when $A1 := A2$ fails, where $A1$ and $A2$ are numerical expressions.

3.5.4 Arithmetic in PROLOG

Although PROLOG was mainly designed as a symbolic programming language, it embodies a number of functions which are used for arithmetic operations. These functions, which are defined in most editions of PROLOG, are:

+	addition
-	subtraction
*	multiplication
/	division
div	integer division
mod	the remainder of integer division

There are also specialized built-in predicates which control **inequality** between numerical expressions.

>	greater
<	smaller
=>	greater or equal
=<	smaller or equal

3.5.5 Type Checking of Objects:

Predicates **var**, **nonvar**, **integer**, **atom**, **atomic**

“**var**” With the predicate **var**(X) we check whether X is a variable which has NOT been updated. If X is a variable which has not been updated, the predicate succeeds, whereas in any other case it fails.

“**nonvar**” With the introduction of the predicate **nonvar**(X) we check whether X is any PROLOG object other than a variable or an updated variable.

“**integer**” The predicate **integer**(X) succeeds if X is an integer. For example the conjunctive query:

? **integer**(I), I is $3 + 5$.

will fail, whereas:

? I is $3 + 5$, **integer**(I).

will succeed and PROLOG will reply $I = 8$. This occurs because before “**integer**(I)” was executed, I had already been updated with the result of the addition $3 + 5$.

“**real**” The predicate **real**(X) succeeds if X is a real number.

“**atom**” The predicate **atom**(X) succeeds only if X is an atomic term.

“**atomic**” The predicate **atomic**(X) succeeds only if X is an atomic term or a number.

3.5.6 The Operators

Frequently, in order to facilitate the input and reading of complex predicates, subsection 3.3.5, we use predicates and functions as **operators**. So, for example, for addition we write $+(a, b)$, where “+” is an operator.

To define an operator completely, we must clearly state its priority, its position and associativity, i.e., its relation with its operands. The complete definition of an operand is achieved by the use of the built-in predicate **op**, and by the introduction of a simple clause of the form:

:- (\langle priority \rangle , \langle position \rangle , [operators name list]).

The programming clauses which define operators are stated in the beginning of the program.

The **priority** of an operator states the order in which each operator will be applied in complex predicates, where there is more than one operator. The priority is a natural number whose range of values depends on the current language edition. For example, in TURBO-PROLOG the priority values are between 1 and 2000. Operators with priority value closer to 1 have higher priority compared to operators with priority value close to the upper priority limit.

For example, we want the meaning of operation:

$$8 + 2 * 2$$

to be:

$$8 + (2 * 2)$$

yielding “12”, and not “ $(8 + 2) * 2$ ” yielding “20”. Therefore the priority of “ $*$ ” is numerically smaller than the priority of “ $+$ ”.

The **position** of an operator denotes the position in which the operator must appear relative to its operands. There are three possibilities for the position of an operator: **before**, **between** or **after** its operands. Therefore, if we want to assign to the predicate `father_of(a,b)` the meaning “*a* is the father of *b*”, as an operator, then naturally we want the predicate to be between its operands, which means, “*a father_of b*”. In this case we have an **infix** operator and its position is denoted by the expression:

$$xfx$$

Therefore, if we give priority 200 to the predicate `father_of`, then we have to introduce the following programming clause:

$$:- \text{op}(200, xfx, \text{father_of}).$$

Similarly, we also have **prefix** and **postfix** operators. Hence, if we define the operator “ \neg ” to express the negation of facts, then it is natural to demand to have it appear before its operand, that is to have it be a prefix operator. If we give it priority 500, then we can define it by the programming clause:

$$:- \text{op}(500, fx, \neg).$$

Now, if we want to define the operator “!” to represent “factorial”, then it would be natural to have “!” appear after its operand, “ $x!$ ”. Therefore “!” is a postfix operator. If we give it priority 400, then it can be defined by the clause:

$$:- \text{op}(400, xf, !).$$

The **associativity** of operators has to do with whether the operator will have lower, higher or equal priority compared with its operands. The explicit declaration of priorities is important when the **operand priority** of an operator is not clearly defined. For example, the arithmetic expression:

$$a - b - c$$

means “ $(a - b) - c$ ” and not “ $a - (b - c)$ ”.

The declaration of operator associativity is done by the separation of the symbols “ x ” and “ y ” which correspond to the operands of the operator. Therefore, with “ fx ” or “ xf ”, that is with the use of variable x , we declare that the operand of the operator has **strictly lower** priority than the operator. With declarations “ fy ” and “ yf ”, that is with the use of variable y , we declare that the priority of the operand is lower than or equal to that of the operator. Of course we can also have a mixture of priorities between operands, as in the declaration “ xfy ”.

In most PROLOG editions, predicates for arithmetic operations and equality are built-in, defined as operators; with corresponding declarations for their priority, their position and their associativity.

For example,

$$:- \text{op}(500, yfx, [+ , -]). \quad (1)$$

$$:- \text{op}(400, yfx, [* , /]). \quad (2)$$

$$:- \text{op}(700, xfx, [=, is, <, >, =<, =>, ==, =\=, \=, :=]).$$

Clause (2) declares that “*”, multiplication, and “/”, division, are operators which have two operands and priority 400, whereas (1) declares that addition and subtraction have two operands and priority 500. Therefore, in arithmetic expressions with “=”, “-”, “*” and “/”, because of their lower priority, “*” and “/” are executed first, and then “+” and “-” are executed.

If we want to define the logical connectives “ \leftrightarrow ”, “ \vee ”, “ \wedge ”, and “ \neg ”, for the formulation of Predicate and Propositional Logic clauses, we define the corresponding operators:

```

:- op(800,xfx,↔).
:- op(700,xfy,∨).
:- op(600,xfy,∧).
:- op(500,fy,¬).

```

Hence, we can introduce the logical equivalence:

$$\neg(A \wedge B) \leftrightarrow \neg A \vee \neg B$$

directly in a PROLOG program as a fact.

3.5.7 *The Towers of Hanoi*

The game of the towers of Hanoi is a typical example of recursive definition. The game is played as follows.

There are three posts, the left, the middle and the right ones, and N disks of different sizes having a hole in the middle. At the initial state, all the disks are on the left post in order of increasing size. The disk at the bottom is the largest one. The goal, i.e., the final state, is to have all the disks transferred in the same size ordering to the right post, as shown in the illustration.

The acceptable disk transfer movements have to obey the rules:

- (a) We move only one disk at a time
- (b) We never place a bigger disk on top of a smaller one.

Obviously, we have to construct a program which finds a path from the initial to the final state, obeying the restrictions in movements. We observe that:

- (1) In the final state, there are no disks on the left post.
- (2) We can use the right post as an auxiliary one and move $N - 1$ disks from the left to the middle one, always according to rules (a) and (b). This will be exactly the recursive step. Then we will have to move the final and largest disk from the left post to the right post.
- (3) Using the left post now as the auxiliary one, we move the $N - 1$ disks from the middle post to the right one, according to rules (a) and (b). This way, we have solved the problem.

Based on the above observations, we construct the following program:

```

move(0, _ , _ , _ ):- !.                                /* 1 */
move(N, X, Y, Z):- N1 is N - 1,
                    move(N1, X, Y, Z),
                    print_move(X, Y),
                    move(N1, Y, Z, X).                    /* 2 */
print_move(X, Y):- write('move a disk
                        from X to Y socket').              /* 3 */
hanoi(N):- move(N, left, middle, right).                    /* 4 */

```

Programming clauses `/* 1 */` and `/* 2 */` are the implementation of the above observations. Clause `/* 3 */` prints out the corresponding movement using the predicate `write`. Clause `/* 4 */` calls the program with the requested number of disks.

Tracing the state space of the program, even for small values of N , is complex. In the general case of N disks, at least $2^N - 1$ movements are needed. The reader can trace the state space of the program for $N = 3$ as an exercise.

3.6 Negation in PROLOG

The language of Predicate Logic, with the logical connectives and the quantifiers which it contains, can formalize most of the colloquial phrases. For example, phrase P of the spoken language

P : “Every canary which is not sick, flies”

is symbolized in the Predicate Logic language with the phrase:

$$P(x) : (\forall x) [(canary(x) \wedge \neg sick(x)) \rightarrow flies(x)]$$

We therefore used the logical connective “ \neg ” to express the negation of the predicate “sick”. However the PROLOG language, based on Horn clauses, cannot express the negation directly, although PROLOG can answer with a “no” in queries. Hence, the formulation, the interpretation, and in general the handling of negation of clauses in PROLOG, are complex matters. The problems which arise are divided into three categories:

- (1) How the “no” answer is interpreted.
- (2) How PROLOG deduces negation of clauses.
- (3) How we use negation in PROLOG.

3.6.1 The Closed World Assumption and Negation by Failure

According to the Closed World Assumption, CWA for short, subsection 3.1.3, if a statement A is neither directly nor indirectly expressed in a program P , that is, if A is neither a fact nor a conclusion based on the data of P , then we accept that its negation holds. Let us consider the following program:

```
perfect_square(4).
perfect_square(9).
```

and the following query is set:

```
? perfect_square(16).
```

The reply of PROLOG will be “no”. This PROLOG reply should not be interpreted as “16 is not a perfect square”, where “not” denotes the standard negation in Mathematical Logic. What is actually meant by the “no” answer is that having checked all the data of this program, the program cannot consider 16 as a perfect square. This “thinking” mechanism of PROLOG and of Logic Programming is based on the Closed World Assumption, and is called **Negation by Failure**.

Actually, we are not able to express negative knowledge by Horn clauses: as defined, a program in Logic Programming contains exclusively Horn clauses which are able to express only positive knowledge, i.e., a program consists of facts, rules and queries in which no negation occurs. Applying resolution for a query Q , we actually prove that $\{\text{facts, rules, } \neg Q\} \vdash \square$, i.e.,

$$\{\text{facts, rules}\} \vdash \neg Q \rightarrow \square, Q$$

This means that in general we cannot prove that a formula of the kind $\neg A$, A without negation, is a consequence of the facts and the rules of some program. There are attempts to expand the methods of Logic Programming to formulae more complicated than Horn clauses in order to increase the expressive power of programs in Logic Programming, [Shep88].

3.6.2 Normal Goals

Consider the following program P :

<code>likes(john,mary).</code>	<code>/* 1 */</code>
<code>likes(john,apples).</code>	<code>/* 2 */</code>
<code>eats(X,Y):- likes(X,Y), edible(Y).</code>	<code>/* 3 */</code>
<code>edible(apples).</code>	<code>/* 4 */</code>

If we ask P whether Mary is edible (!).

`? edible(mary).`

the answer will be “no”, i.e., $\neg \text{edible(mary)}$, because of the CWA. Then it is not meaningless to ask P :

<code>? ¬edible(mary).</code>	<code>/* 5 */</code>
-------------------------------	----------------------

To answer this query, the program will check whether the goal

? edible(mary).

succeeds. Since “edible(mary).” does not match any data of P , it does not succeed. Therefore, the answer to $/ * 5 * /$ will be “no”.

Goals of the kind “? $\neg A$.” are called **normal goals**, [Lloy87, NiMa95]. The matching procedure which takes place during the execution of programs allows the definition of success and failure of normal goals:

When the goal “? A .” succeeds, then the goal “? $\neg A$.” fails, i.e., A is a consequence of the program being considered.

When “? A .” fails, then the goal “? $\neg A$.” succeeds, i.e., $\neg A$ is a consequence of the program being considered.

Hence, negation in normal goals is also characterized by the failure or success of the corresponding unnegated goal, therefore it is called **negation by failure**, NF for short. The relevant (meta)rule is:

$$\frac{? \neg A., \quad ? A. \text{ fails}}{\square} \quad \text{NF}$$

This means that when “? A .” fails in the program P , then

$$\neg \neg A \rightarrow \square$$

i.e., $\neg A$ is a consequence of P .

Obviously we have to deal with two kinds of negation here, one being the standard logical connective, and the other the procedural one declared by the statement “? A . fails”. Let us denote this procedural negation by \sim ; then the fact that ? A . fails can be denoted by $\sim A$. Informally, “? $\neg A$.” can be considered as being equivalent to A . *The rule NF actually allows the resolution between ? $\neg A$. (i.e., A) and $\sim A$!* This kind of resolution is called **NF-resolution**.

The attempt to give a theoretical proof of the validity of the NF rule and to bring closer the logical negation and the procedural negation, has led to the notion of the completion of programs [Clar78, Lloy87, NiMa95]. The principal idea is to consider the predicates occurring in some program P by means of : , i.e., \leftarrow , as *completely* defined by means of \leftrightarrow .

3.6.3 Completion of Programs

The logical basis for the completion of a PROLOG program is given by the following axioms (see Remark 2.3.7), universally quantified:

- | | |
|--|---|
| $(A_6) \quad x = x$
$(A_7) \quad x = y \rightarrow (A \leftrightarrow A_1)$ | axiom of equality of terms
rule of substitution of equal terms |
|--|---|

the following valid formulae of PrL:

- (1) $(B \rightarrow A) \wedge (C \rightarrow A) \leftrightarrow (B \vee C) \rightarrow A$
- (2) $(A(x, y) \rightarrow B(x, y)) \leftrightarrow [(\exists x)(\exists y)(A(x, y) \wedge x = a \wedge y = b) \rightarrow B(a, b)]$
- (3) $A(a, b) \leftrightarrow (x = a \wedge y = b \rightarrow A(x, y))$

and some axioms concerning the properties of the functions appearing in the programs. Let $x \neq y$ denote $\neg(x = y)$. All functions which occur in the program to be completed, [Clar78, Lloy87, NiMa95], must obey the following specific, i.e., non-logical axioms, universally quantified:

- (F₁) $f(x_1, \dots, x_n) \neq g(y_1, \dots, y_m)$ for all functions f, g for which $f \neq g$
- (F₂) $f(x_1, \dots, x_n) = f(y_1, \dots, y_n) \rightarrow (x_1 = y_1) \wedge \dots \wedge (x_n = y_n)$
- (F₃) $f(x) \neq x$ for all x proper subterms of $f(x)$

i.e., different functions have different values, a function has different values for different variables, and, the value of a function cannot be a variable occurring in the function, [Clar78].

Let us find the completion for the example of the previous subsection. To make the intuition behind the completion more obvious, we will use \leftarrow instead of $:-$.

Applying the formulae (2) and (3) to the clauses of P , we obtain the following equivalent form of P :

```
likes(t1, t2) ← (t1 = mary ∧ t2 = john)
likes(t1, t2) ← (t1 = apples ∧ t2 = john)
eats(t3, t4) ← (∃Y)(∃X)(t3 = Y ∧ t4 = X ∧ likes(X, Y) ∧ edible(X))
edible(t5) ← t5 = apples
```


and by means of the formula (1) we have:

$$\begin{aligned} \text{likes}(t_1, t_2) &\leftarrow (t_1 = \text{mary} \wedge t_2 = \text{john}) \vee (t_1 = \text{apples} \wedge t_2 = \text{john}) \\ \text{eats}(t_3, t_4) &\leftarrow (\exists Y)(\exists X)(t_3 = Y \wedge t_4 = X \wedge \text{likes}(X, Y) \wedge \text{edible}(X)) \\ \text{edible}(t_5) &\leftarrow t_5 = \text{apples} \end{aligned}$$

Now we regard the formulae at the left-hand side of \leftarrow as defining *completely* the predicates “likes”, “eats” and “edible”: according to the CWA, the right-hand side of the above formulae is the only available information on “likes”, “eats” and “edible” in P , hence it must definitely define these predicates. This means that \leftarrow can be considered as equivalence, \leftrightarrow ! Consequently, P must have the following *complete* form:

$$\begin{aligned} \text{likes}(t_1, t_2) &\leftrightarrow (t_1 = \text{mary} \wedge t_2 = \text{john}) \vee (t_1 = \text{apples} \wedge t_2 = \text{john}) \\ \text{eats}(t_3, t_4) &\leftrightarrow (\exists Y)(\exists X)(t_3 = Y \wedge t_4 = X \wedge \text{likes}(X, Y) \wedge \text{edible}(X)) \\ \text{edible}(t_5) &\leftrightarrow t_5 = \text{apples} \end{aligned}$$

This way we are able to immediately define the negation of these predicates, e.g.,

$$\neg \text{edible}(t_5) \leftrightarrow t_5 \neq \text{apples}$$

Hence, negated queries and normal goals are meaningful this way, and, moreover, negation by failure is closer to logical negation. Nevertheless, the interpretation of an implication as an equivalence cannot be justified within the framework of PrL, and it is meaningful only as a semantic interpretation of programs.

Formally:

Definition 3.6.3.1:

- (i) If the predicate A occurring in a program P appears in the head of the following Horn clauses of P :

$$\begin{aligned} A(x_{1_1}, \dots, x_{n_1}) &\leftarrow W_1 \\ &\cdot \\ &\cdot \\ &\cdot \\ A(x_{1_k}, \dots, x_{n_k}) &\leftarrow W_k \end{aligned}$$

then the **complete definition** of A in P is

$$(\forall t_1) \cdots (\forall t_n) (A(t_1, \dots, t_n) \leftrightarrow E_1 \vee \cdots \vee E_k)$$

for

$$E_i := (\exists y_1) \cdots (\exists y_d) (t_1 = x_{1_i} \wedge \cdots \wedge t_n = x_{n_i} \wedge W_i)$$

y_1, \dots, y_d all variables occurring in the clause $A(x_{1_i}, \dots, x_{n_i}) \leftarrow W_i$ and t_1, \dots, t_n are new variables, i.e., they do not occur in P .

- (ii) If there is no occurrence of A in the left-hand side of any clause of P , the **complete definition** of A in P is

$$(\forall t_1) \cdots (\forall t_n) \neg A(t_1, \dots, t_n)$$

i.e., $(\forall t_1) \cdots (\forall t_n) A(t_1, \dots, t_n) \leftrightarrow \square$.

- (iii) The **completion** of P , $\text{compl}(P)$, is the set of the complete definitions of all predicates occurring in P . ■ 3.6.3.1

The benefits of using $\text{compl}(P)$ consist of the theoretical means allowing the increase of the expressive power of P without losing any consequence of P , and of a kind of legitimacy for the rule NF. The relevant theorems, whose proofs can be found, for example, in [NiMa95, Lloy87], are:

Theorem 3.6.3.2:

If $P \models A$ then $\text{compl}(P) \models A$.

■ 3.6.3.2

Theorem 3.6.3.3:

The NF rule is sound for $\text{compl}(P)$.

■ 3.6.3.3

In general, the procedures in Logic Programming do not allow us to find the values x satisfying queries of the form “ $? \neg A(x)$.”, and it is often assumed that the NF rule applies only to ground instances, [NiMa95]. If a normal goal “ $\leftarrow \neg A$ ” contains only ground instances, then the corresponding negation is called **safe**, otherwise it is called **unsafe**. Many Logic Programming languages allow only safe negation and incorporate mechanisms to delay the processing of normal goals until the occurring variables take a concrete value.

3.6.4 Normal Programs and Stratification

Nevertheless, goals with negated predicates are allowed in Logic Programming and PROLOG. If we allow negations to occur in the body of rules, the relevant programs are called **normal** or **general programs**, [NiMa95, Lloy87]. However, the situation then becomes more complicated: a normal program P may contain a clause of the form:

$$A :- \neg A$$

i.e., $A \leftarrow \neg A$, which is equivalent to A ; but in $\text{compl}(P)$, it takes the form:

$$A \leftrightarrow \neg A$$

equivalent to $A \wedge \neg A$! Thus $\text{compl}(P)$ is not a consistent set of sentences in this case!

Nevertheless, normal programs which are *stratified* do have consistent completions. Let us consider the following program P :

$$\begin{array}{ll} A :- \neg B. & /* \ 1 \ */ \\ A :- B, C. & /* \ 2 \ */ \\ C. & /* \ 3 \ */ \end{array}$$

Then $\text{compl}(P) = \{A \leftrightarrow \neg B \vee C\}$, which is obviously a consistent set.

We can clarify what we mean by a program being stratified as follows. Notice that in P , A is determined both by $\neg B$ and by $B \wedge C$, i.e., by both B and C . Roughly speaking, what we would like to do is to separate A , B and C into a hierarchy of levels of definition, with predicates in the higher levels being defined by those in the lower levels. Certainly the highest “definition level” must contain A , since A depends on B and C . So B must be contained in a lower “definition level” than that of A , since B would have to have been determined already in order to be able to define A .

Since these “definition levels” must completely determine the predicates in the upper levels, they have to be disjoint. Actually, we prefer to refine the approach we have just described to allow a predicate X to have the same level as a predicate Y it defines, provided it was X and not its negation which was used to define Y . In our example, the predicates occurring in P can be partitioned into the two

“definition levels” $S_0 = \{B\}$ and $S_1 = \{A, C\}$, such that the negated predicate B which determines the predicate A has been determined at a level lower than that at which A is determined. Since it was not the negation of C , but C itself which was used to determine A , the level S_1 may contain both A and C .

The reason we want to allow this approach, is that stratification is designed to control unwanted effects of recursion. In general, a negated predicate D which determines a predicate F at the left-hand side of $:-$ must have been determined at a lower level, in order to avoid recursions during the backtracking procedure which lead to formulae of the kind $D:- \neg D$, as for example in the following program:

$$A:- B.$$

$$B:- \neg A.$$

Formally, [NiMa95, Lloy97]:

Definition 3.6.4.1:

A normal program P is called **stratified** if the set of the predicates which occur in P can be partitioned into S_0, \dots, S_n such that if $A:- B_1, \dots, B_m \in P$ and $A \in S_k$, then, if in B_i , $1 \leq i \leq m$, occurs no negation, then

$$B_i \in S_0 \cup S_1 \cup \dots \cup S_k$$

while, if B_i is of the form $\neg C_i$, then

$$C_i \in S_0 \cup S_1 \cup \dots \cup S_{k-1}$$

■ 3.6.4.1

Theorem 3.6.4.2:

If P is stratified, then $\text{compl}(P)$ is consistent.

■ 3.6.4.2

Theorem 3.6.4.3:

If P is a normal program, then the NF rule is sound for $\text{compl}(P)$.

■ 3.6.4.3

Completeness results concerning the NF rule are beyond the scope of this book and can be found in [Lloy87, NiMa95].

Let us now see an example of the application of the NF rule to a normal program, i.e., to a program expressing negative knowledge. Suppose we have a program P with data expressing negative knowledge:

```
interested_in(X,Y):- ¬unsuited(X,Y).      /* 1 */
unsuited(X,Y):- ¬likes(X,Y).              /* 2 */
likes(mary,john).                          /* 3 */
```

and we want to know whether Mary is interested in John,

```
? interested_in(mary,john).               /* 4 */
```

The program will proceed as follows: /* 4 */ matches the left-hand side of /* 1 */ for $X = \text{mary}$ and $Y = \text{john}$. The next goal is:

```
? ¬unsuited(mary,john).                   /* 5 */
```

The algorithm checks the goal:

```
? unsuited(mary,john).                     /* 6 */
```

/* 6 */ matches the left-hand side of /* 2 */ for $X = \text{mary}$ and $Y = \text{john}$. The next goal is:

```
? ¬likes(mary,john).                       /* 7 */
```

The algorithm checks the goal:

```
? likes(mary,john).
```

which succeeds by /* 3 */. By NF, the goal /* 7 */ fails, i.e., the goal /* 6 */ also fails, and, consequently, the goal /* 5 */ succeeds and the answer to the query /* 4 */ is “yes”!

On the other hand, if we try to apply resolution to the corresponding logic sentences, we cannot obtain results easily, because P does not consist of Horn clauses. Let us consider the following equivalent form of P :

- interested_in(X, Y) \vee unsuited(X, Y) (1)
- unsuited(X, Y) \vee likes(X, Y) (2)
- likes(mary,john) (3)
- ¬interested_in(mary,john) (4)

In this form of P , we can only apply resolution between (1) and (4). The sole result which can be obtained is “unsuited(mary, john)”. On the contrary, applying the metarule NF, we are able to find an acceptable solution which is correct under the hypothesis that the data which completely characterizes the predicates in P is the one given in P !

3.6.5 The Predicate fail

A given goal in PROLOG fails, which means that it takes a false truth-value, if all the possible attempts to satisfy it through backtracking fail. Failure, and consequently negation, can be declared in PROLOG through a special built-in predicate which expresses **failure**, namely **fail**, which has a false truth-value and always fails. So

$$\text{fail} \leftrightarrow Q \wedge \neg Q$$

where Q is any formula of Predicate Logic.

Let us take as an example the sentence:

$$A : \quad \text{“John is not a canary”}$$

Sentence A can be expressed in PROLOG by the rule:

$$\text{canary(john)} :- \text{fail.} \quad (*)$$

meaning that “the goal **canary(john)** fails”. Let us assume that the database of a PROLOG program contains only the above rule (*). Then the query:

$$? \text{ canary(john).}$$

will fail. After the unification of the query with the head of rule (*), the new sub-goal **fail** fails, by means of the definition of **fail**. This is exactly the procedural interpretation of **fail**: the head of the rule, in whose body **fail** is found, fails.

Here we have to emphasize that clauses of the form:

$$A :- \text{fail.}$$

are logically true clauses of Predicate Logic. The clause

$$Q \wedge \neg Q \rightarrow A$$

is true in every interpretation of Predicate Logic (Definitions 2.5.5, 2.5.19, and the proof of Corollary 1.5.4).

In general, negation in PROLOG can be expressed using a specific combination of cut and failure: let us take for example the clause

$$P(x) : (\forall x) [(canary(x) \wedge \neg sick(x)) \rightarrow flies(x)]$$

on page 262. Assume that we want to express the relation between flying objects and canaries. Depending on the case, we will have canaries which fly, but also canaries which do not fly, for example, because they are wounded. We therefore have to include in the program both possibilities, i.e., both the canaries which do not fly because they are sick and the canaries which fly. This can be done with the PROLOG program:

```
flies(X):- sick(X), !, fail.           /* 1 */
flies(X):- canary(X).                 /* 2 */
```

In this program we use the cut, `!`, to avoid the activation of the second rule in the case of a given canary X which is sick. If a canary X is sick, then the cut will prevent PROLOG from backtracking to the second rule and `fail` will lead to the failure of the head of rule `/* 1 */`. Hence, we will conclude that this given canary X does not fly. If a canary is not sick, or it is not specifically mentioned in the data base that the canary is sick, then the first rule will fail and PROLOG will backtrack (the cut is not executed), activating the second rule, whose head will be satisfied.

We therefore see that negation in PROLOG can be expressed by the use of a **cut-failure combination**.

3.6.6 The Predicate `not`

A different attempt to express negation in PROLOG is the special predicate `not(Q)`, subsection 3.3.5, which denotes the negation of Q , and is satisfied when the goal Q fails. The predicate `not` essentially simulates the procedural operation of the cut-failure combination and can be defined by the following programming clauses:

```

not(Q) :- Q, !, fail.           /* 3 */
not(Q).                         /* 4 */

```

Therefore, if we ask:

```
? not(Q).
```

PROLOG will activate rule /* 3 */. If Q is not explicitly defined in the database or if we cannot conclude Q from the data of the program, then the subgoal Q in the body of the rule will fail. Next, PROLOG will attempt to satisfy $\text{not}(Q)$ through the activation of the clause /* 4 */. Since Q fails, $\text{not}(Q)$ is satisfied by the NF rule, and PROLOG will reply “yes”, i.e., the negation of Q holds. Now, if either Q is explicitly defined, or we can infer Q from the data, the subgoals Q , $!$, and **fail** in rule /* 3 */ will be satisfied in turn. The cut, $!$, prevents the activation of clause /* 4 */ and **fail** denotes the failure of $\text{not}(Q)$. Therefore, the head of rule /* 3 */, $\text{not}(Q)$, does not succeed, and PROLOG will reply with “no”, which means that Q can be satisfied, whereas $\text{not}(Q)$ fails.

The use of **not** may help a lot expressing negations in programs. In the example of the previous subsection, clauses /* 1 */ and /* 2 */ can be represented by the rule:

```
flies(X) :- canary(X), not(sick(X)).    /* 1' */
```

The predicate **not** is built-in in most PROLOG editions. If it is not, then we can define **not** by programming clauses /* 3 */ and /* 4 */. Once again, we observe that PROLOG offers the ability to define the predicates of its language through the language itself (PROLOG through PROLOG).

3.6.7 Nonmonotonic Logics

Reasoning based on the CWA, presents inherent weaknesses. Let us for example assume the following singleton:

$$S = \{\sigma \vee \varphi\}$$

where σ and φ are clauses in Predicate Logic. If we represent by $S \models_{\text{CWA}} A$ the fact that clause A is a consequence of S under the Closed World Assumption, then:

$$S \models_{\text{CWA}} \neg\sigma \quad \text{and} \quad S \models_{\text{CWA}} \neg\varphi$$

because σ and φ are neither explicitly defined, nor can be deduced from the data of set S . Therefore, the consequences of S under the CWA, $\text{Con}_{\text{CWA}}(S)$, are:

$$\text{Con}_{\text{CWA}}(S) = \{\sigma \vee \varphi, \neg\sigma, \neg\varphi\}$$

which is obviously inconsistent (for example, resolution two times will result in \square). Therefore, the use of the Closed World Assumption leads to inconsistencies, even in trivial cases.

One further problem which arises when working with the CWA has to do with the generality of the rules of a program and their exceptions. For example, according to rule / * 1' * / of subsection 3.6.2, to conclude that something flies, we have to eliminate the possibility that it is sick. This means that we have to **explicitly** enter all the exceptions to rule / * 1' * /. This in general is impossible, because the exceptions might not be known in advance, but might arise during the execution of the program.

In recent years, there have been attempts to develop formal methods for conclusion deduction which do not present problems similar to those presented by the CWA. The related research aims at the development of new logics, called **Nonmonotonic Logics**. The main difference between Nonmonotonic Logics and the classical Predicate and Propositional Logics is that in Nonmonotonic Logics we can invalidate previous conclusions when we insert new conclusions. Typically, in the framework of classical Logic, the following holds:

$$\text{If } S \models A \text{ then it follows that } S \cup S' \models A$$

meaning that if clause A is a consequence of a set S of clauses, then A remains a consequence of any extension of S by a clause set S' . In Nonmonotonic Logics, NM for short, it holds that:

$$\text{If } S \models_{\text{NM}} A \text{ then in general } S \cup S' \not\models_{\text{NM}} A$$

which means that if clause A is a consequence of S in a Nonmonotonic Logic, then A in general does not remain a consequence of the extension of S by some clause set S' .

Cancellation of previous conclusions after the expansion of data is closer to the natural way that people think, i.e., revising their beliefs according to the information which they have about their surroundings. Nonmonotonic Logics aspire to become the new foundation of Logic Programming languages, like PROLOG, aiming at the development of “smarter” and more capable systems. In Nonmonotonic Logics, the universe is not closed:

The universe is neither static, nor once and for all given, but it constantly expands by the inflow of new data and information; whereas the concept of truth in the universe is relative, and always depends on the knowledge which the universe possesses at each moment.

The inference mechanism in Nonmonotonic Logics is based on the Closed World Assumption at each time instant:

Assume that the universe is expressed by the set S_1 of clauses at time instant t_1 ; whereas at time instant t_2 , with $t_2 > t_1$, it is expressed by the set S_2 , where $S_1 \neq S_2$.

In order to find the consequences of S_1 and S_2 , in other words, in order to answer queries concerning S_1 or S_2 , we assume that the Closed World Assumption holds in both S_1 and S_2 . This means that at time instant t_1 , the universe is closed and is expressed by the set of clauses S_1 ; and at time instant t_2 , the universe is also closed and is expressed by the set S_2 .

More information about Nonmonotonic Logics can be found in, for example, [Turn84] and [Besn89].

3.7 Expert Systems

3.7.1 *Artificial Intelligence*

The field of **Artificial Intelligence**, AI for short, deals with the development and formalization of intelligent methods and solution procedures for problems. By the term “intelligent”, we mean methods which allow reasoning and judgement in a way comparable to the human thinking process. Research in AI aims at the construction of symbolic representations of the world, and covers areas such as robotics, natural language understanding, and expert systems.

3.7.2 Expert Systems and Knowledge Management

Expert systems are computer programs capable of solving complex problems which require extensive knowledge and experience in order to be solved [BuSh84].

The greatest difference between the traditional programs and expert systems, is that traditional programs manage data whereas expert systems manage knowledge. The differences between the two different kinds of programming can be presented in detail in the following table:

Data Management	Knowledge Management
Data Representation and Use	Knowledge Representation and Use
Algorithmic Procedures	Search Procedures
Repeating Procedures	Conclusive Procedures

Expert systems are specialized systems, meaning that they refer to subfields of specialized application fields; such as medical diagnosis systems, mechanical failure diagnosis systems, systems for the synthesis of complex chemical compounds, etc..

The means by which expert systems achieve their goals are based on sets of **facts** and sets of **heuristics**, i.e., rules for knowledge management. These facts and rules are developed by specialized scientists who work in the field to which the expert system has applications. We can say that this collection of facts and heuristics constitutes the knowledge of the system in this specific field, and hence expert systems are often called **Knowledge Based Systems**.

An expert system consists of a **database**, an **inference mechanism**, and the **interface** for the communication between the system and the user.

The **database** is a simple set of elements and states, which constitutes the description of the universe of the given field.

Knowledge representation [Watt90] in systems based on rules is achieved by using rules of the form:

```
IF <condition.1> and ... <condition.k>
THEN <action>
```

The **inference mechanism** consists of a set of general purpose rules, which is used to guide and control the inference process. This mechanism implements

general models, which characterize and formalize processing and resolution procedures of problems; such as the depth-first search, the backtracking and other general purpose inference procedures described in previous sections.

In cases where the inference mechanism can become autonomous with respect to the specialized knowledge which a given expert system contains, and work independently, it can be used for the development of new expert systems. Such systems with general inference power are called **shells**.

The **interface** is responsible for communication between the user and the system. In most cases, specialized subsystems which analyze and synthesize natural language (parsers) are used, in order to get a natural and friendly communication between the user and the system.

The acquisition and formalization of knowledge needed to be embodied in a given expert system, is one of the most fundamental and, at the same time, difficult stages during the development of the system. Indeed, it is called a “knowledge acquisition bottleneck”. The difficulty lies in the provision and simultaneous formalization of specific rules which express the procedures which a specialist in problem solution in the field follows. Therefore, the work of the **knowledge engineer**, in other words the person whose duty is to find and express these procedures, is the basic element in the successful development and use of an expert system.

In recent years, due to the importance of knowledge acquisition mechanisms, **automatic knowledge acquisition systems** have been developed. The methods used are in many cases very successful, and present great theoretic interest [OlRu87, Quin86].

The **tools** used in the development of expert systems are divided into two main categories [WeKu84]: the **high level symbol processing languages**, and the **general purpose expert systems** or **shells**. The most widespread high level symbol processing languages are LISP [WiBe89] and PROLOG.

The shells are used to facilitate and accelerate the development of special expert systems. With the use of special user friendly interfaces and the structures which they embody, they facilitate both the input and the revision, if needed, of an expert system's facts and rules. Furthermore, they embody an autonomous inference procedure capable of transacting and deriving conclusions from different databases. This offers the ability to develop expert system prototypes rapidly.

3.7.3 An Expert-System for Kidney Diseases

The **Kidney Expert System**, KES, is a system which diagnoses kidney diseases, and was developed with the cooperation of the Logic and Logic Programming Group of the department of Mathematics in the University of Patras, and a team of specialized doctors from the University Hospital of Patras.

In the presentation which follows, the clauses and procedures which implement KES have been limited only to those clauses and procedures which describe the structure and operation of the system, and at the same time display the use of PROLOG in the development of expert systems.

The rules which constitute the special purpose Knowledge Base of KES are the formalization of the manner in which the doctor makes a diagnosis based on the symptoms of the patient: the fact that the patient is relatively old, that his urine presents a qualitative change as well as his blood, and that the kidneys appear swollen. If all these conditions are met, then the patient probably suffers from hydronephrosis. Therefore, the rules of KES are of the following general form:

```

IF          the data of the patient satisfy some given requirements
    and      reported(X)
    and/or   observed⟨symptom_1⟩
    and/or   .....
.....
    and/or   observed⟨symptom_k⟩
    and/or   found⟨lab_result_1⟩
.....
    and/or   found⟨lab_result_n⟩
THEN        the diagnosis is ⟨disease⟩ with probability P

```

These rules were formed under the guidance of specialized doctors. Therefore, for hydronephrosis, the corresponding rule is:

```

diagnose(hydronephrosis,0.7):- age(H), H > 11,
                                reported(change_in_quality_of_urine),
                                observed(bloodurine),
                                observed(twosided_kidney_expansion).

```

where 0.7 is the probability that the given patient has hydronephrosis. The theoretic probability measure which is appended to each diagnosis is an *a priori* assessment of probability. Therefore, the possibility of diagnosing the same disease through different paths and with different probability statements is offered.

The basic predicates included in KES are:

data : Accepts four arguments: sex, age, first name, and last name of the patient.

observed : Accepts one argument: the name of the symptom observed by the examiner.

found : Accepts one argument: the name of the result of a possible laboratory test.

reported : Accepts one argument: one of `pain`, `change_in_quality_of_urine`, `change_in_quantity_of_urine`, `change_in_frequency_of_urination`. The predicate `reported` is used for the grouping of symptoms. Therefore a rule of the above general format which does not contain in its body the predicate `reported`, with any argument, will not be activated, resulting in the acceleration of the inference procedure.

diagnose : Accepts two arguments: the name of the diagnosis and the corresponding probability that the patient has the disease stated by `diagnose`.

After the description of the basic predicates and the special rule of KES, we can move on to the presentation of the procedures and the corresponding programming clauses which implement the inference mechanism of the system.

At the beginning, we have to define the input procedure of the patient's data, in other words, predicate `data`:

```
data(S, A, L, F):- write('Patient's sex(m/f)'), read(S), nl,
                  write('Patient's age'), read(A), nl,
                  write('Patient's last name'), read(L), nl,
                  write('Patient's first name'), read(F), nl
                  assertz(sex(S)),
                  assertz(age(A)),
                  assertz(surname(L)),
                  assertz(name(F)).
```

Next the input procedure for the symptoms reported by the patient has to be defined, in other words, predicate `reported`:

```
reported(X):- write('Reported from patient', X, '(y/n)?'),
              read(ANSWER), nl,
              ANSWER = 'y',
              assertz(reported(X)).
```

Now, the input procedures for the symptoms observed during the examination, as well as the possible laboratory results which the doctor might have available, have to be defined:

```
observed(S): write('Observed during the examination',S,'(y/n)?'),
             read(ANSWER), nl,
             ANSWER = 'y',
             assertz(observed(S)).

found(R):- write('Laboratory result',R,'observed (y/n)?'),
           read(ANSWER), nl,
           ANSWER = 'y',
           assertz(observed(S)).
```

In the above definitions, the existence of `assertz` results in the insertion of the basic implementation of the head of the corresponding rule, when it is satisfied. Therefore, the interaction between the system and the user during the inference procedure is recorded and stored for any further processing.

Finally we define the general diagnose predicate:

```
proceed.in.diagnose:- !, data(S,A,L,F),
                     diagnose(D,P),
                     assertz(diagnose(D,P)).
```

If the above predicate is set as a query and if we assume that the database incorporates heuristics, then PROLOG will move on to derive conclusions. In reality, there will be a dialogue between the user and PROLOG which will result in the diagnosis which can be derived, depending on the user's responses.

Let us assume for example that the database contains the following diagnose rules:

```
diagnose(acute_pyelonephritis,0.7):-
    reported(pain),
    observed(high_fever_with_shiver),
    observed(bloodurine).

diagnose(acute_pyelonephritis,0.8):-
    age(A), A < 9,
    diagnose(acute_pyelonephritis,0.7),
    found(positive_urine_culture).

diagnose(acute_pyelonephritis,0.9):-
    reported(change_in_frequency_of_urinations),
    reported(pain),
    diagnose(acute_pyelonephritis,0.8),
    observed(pain_in_kidneys),
    observed(micturition),
    observed(dysury).
```

Asking now the query:

? proceed_in_diagnose.

the following dialogue takes place, where the user's responses are underlined:

? Patient's sex(m/f) m

? Patient's age 11

? Patient's last name SURNAME

? Patient's first name NAME

— PROLOG enters the following clauses:

sex(m).

age(11).

surname(SURNAME).

name(NAME).

? Reported from patient pain (y/n)? y

— PROLOG enters the clause:

reported(pain).

? Observed during examination pyuria (y/n)? y

— PROLOG enters the clauses

observed(pyuria).

diagnose(acute_pyelonephritis, 0.7).

and attempts to satisfy the predicate, **diagnose**, in a different way. Since in the terms of the second clause for **diagnose**, **age(A)** is mentioned, which is satisfied for $A = 11$, the term $A < 9$ is not satisfied and therefore PROLOG goes on to examine the third clause for the predicate, **diagnose**.

? Reported from patient change_in_frequency_of_urinations (y/n)? n

— PROLOG abandons this rule as well, and ends the attempt to satisfy the initial query, and then replies because of the cut.

$D = \text{acute_pyelonephritis}$

$P = 0.7$

3.8 The Evolution of Logic Programming

3.8.1 Editions of PROLOG

PROLOG, in the form of PROLOG I, was designed as an artificial language for natural language processing based on logic [Colm90, Col90a]. Despite its great capability in the expression and resolution of complex problems, its abilities in arithmetic calculations were relatively limited.

PROLOG II, the PROLOG which we have described, is a relatively slick and rich language in dealing with arithmetic calculations, due to the addition of infinite trees and the predicate “\=”.

The latest edition of PROLOG, PROLOG III [Colm90, Col90a], has much greater capabilities. It includes a built-in mechanism to manage infinite trees, Boolean Algebra, the predicates

$$< , = < , > , = > \text{ and } \backslash =$$

and the addition, subtraction, and multiplication functions with constants.

PROLOG III is a Constraint Logic Programming language [Cohe90]. Constraint Logic Programming is the result of the attempt to enrich the Horn clause language with variables which take values in different domains, for example, trees, Boolean Algebra, real or rational numbers, etc..

3.8.2 Dialects of PROLOG

PROLOG, actually, offers a mechanism for the execution of logic programs. The dialects of PROLOG use the same basic mechanism, and their difference lies mainly in the manner of interaction with the user.

MICRO-PROLOG [ClMc84], TURBO-PROLOG, ARITY-PROLOG and Δ -PROLOG [Xant90] are dialects for small computers. Δ -PROLOG is the Greek dialect and uses Greek characters.

IC-PROLOG, *mu*-PROLOG and *nu*-PROLOG allow for a more complex selection of predicates for unification [Lloy87]. *Nu*-PROLOG, through its automatic pre-processor, is much more responsive in controlling the program compared with the other dialects of PROLOG, and is considered to be a language very close to Ideal Logic Programming.

3.8.3 PROLOG and Metaprogramming

In chapters 1 and 2, we saw that for a language \mathcal{L} of Predicate or Propositional Logic, there exists a corresponding metalanguage; the language in which we express comments for \mathcal{L} and its clauses, Remark 1.5.2. The same capability also exists in PROLOG. The metaprograms of PROLOG have PROLOG programs as terms, and they examine their relations and properties. For example [Kowa90], the predicate:

$$\text{demo}(T, P)$$

which is interpreted as

“program T is used for the proof of clause P ”

Therefore, using the capabilities of PROLOG for metaprogramming, beginning from simple programs and using metalanguage in every step, we can write a PROLOG program which processes a group of PROLOG programs, each one of which processes other PROLOG programs, etc..

The purpose of using metaprogramming is automatic development of PROLOG programs, and automatic verification of properties of certain PROLOG programs.

3.8.4 PROLOG and Parallelism

The research for the development of faster and more effective computing machines led to the construction of parallel computers. The classic computers process each command or statement of the program one after the other, sequentially; and so the machine is capable of executing only one command or statement in any one computational step. In parallel computers, one computational step might consist of mutually independent commands or statements which are executed concurrently by means of multiprocessors.

Although the use of parallel algorithms and parallel computers gives simple and natural solutions to complex problems, it also creates a number of new problems having to do with the synchronization of the communications between the independent processes, as well as problems having to do with the control of the program. These problems can be overcome by special purpose programming languages which are capable of expressing parallelism, e.g., OCCAM, and which attempt to bridge the gap between this new multiprocessor technology (hardware) and the classical sequential software.

PROLOG is unable to express parallelism through its language. Moreover, as we have seen, a PROLOG program is always executed by sequential processes. Therefore, parallel programming languages are being developed and have been enhanced in recent years. These languages are based on the parallel interpretation of logic programs and contain special elements for the synchronization of processes and the control of the program.

The **Relational Language**, Clark and Gregory (1981), was the first logic programming language which could express nondeterministic choices between independent declarations of the program. The clauses of the language have one of the following forms:

$$A :- G_1, \dots, G_k / B_1, B_2, \dots, B_r \quad (*)$$

$$A :- G_1, \dots, G_k / S_1 \parallel S_2 \parallel \dots \parallel S_r \quad (**)$$

where $A, G_1, \dots, G_k, B_1, \dots, B_r$ are predicates, S_1, \dots, S_r are conjunctions of predicates, A is the head of clauses $(*)$ and $(**)$, B_1, \dots, B_r the tail of $(*)$, S_1, \dots, S_r the tail of $(**)$, and G_1, \dots, G_k the conditions (guards) which have to hold in order for the program to use $(*)$ and $(**)$. The difference between clauses $(*)$ and $(**)$ is procedural: the conjunctions $S_i, i = 1, \dots, r$, are executed by independent processors. The unification without backtracking is executed in parallel.

The Relational Language, the first parallel logic programming language, has relatively limited abilities. Therefore, a Relational Language program can only verify the truth of simple relations between facts.

Concurrent PROLOG, Shapiro (1983), uses programming clauses of the form $(*)$ but two different types of variables; the **read_only variables** and the ordinary variables. The read_only variables cannot be unified. The unification algorithm has to wait until these variables take a certain value. Through this delay the synchronization problem is also solved.

PARLOG (**parallel programming in logic**), Clark and Gregory (1984), is an improvement of the Relational Language, whereas **GHC** (**Guarded Horn Clauses**), Ueda (1985), is a combination of the best elements of PARLOG and concurrent PROLOG.

In general, the field of parallel logic programming languages is still evolving. More information about these languages can be found in [Shap87].

3.9 PROLOG and Predicate Logic

As we have seen, Logic Programming deals with a specific class of Predicate Logic clauses, the Horn clauses. Therefore, the PROLOG language without the special predicate symbol “.”, subsection 3.3.7, is a subset of the language of Predicate Logic. PROLOG, which is actually a conclusion inference mechanism from assumptions within Logic Programming, can be distinguished into two varieties:

- (1) Pure PROLOG
- (2) Real PROLOG

Pure PROLOG does not contain control mechanisms and therefore “cut”, “!”, and predicates **fail** and **not**, for the negation, are not defined in its language. Real PROLOG is the *extension* of pure PROLOG by the predicates “cut”, **fail** and **not**. This division of PROLOG into pure and real is very important; because the completeness theorems of Predicate Logic hold in pure PROLOG, yet they do NOT hold in real PROLOG:

The data of a given PROLOG program are a set of propositions S of Predicate Logic. Every query of the form “? P .” in the program leads to the proof or disproof of $S \vdash_R P$. In *pure* PROLOG we are sure that, because of the completeness theorem (Theorem 2.10.11) and Robinson’s theorem (Theorem 2.9.8), the unification algorithm will stop either after having found *all* the values of variables for which $S \vdash_R P$, or after disproving $S \vdash_R P$.

In *real* PROLOG, the use of “cut” is decisive. By cutting off a subtree of the solution tree with “cut”, we might hinder the program from finding the only values of variables for which $S \vdash_R P$. The example of subsection 3.4.4 (pp. 240-242) is characteristic: The use of rule / * 1' */ forced PROLOG to give a wrong answer.

Therefore, whereas the programs of real PROLOG which use “cut”, **fail**, and **not**, are very slick, the programmer and the user do not have any guarantee that the results of their programs will be correct.

The programs in pure PROLOG are more demanding in terms of execution time. The need for faster programs led to the development of techniques which improved the performance of programs even in real PROLOG. Real PROLOG, without

Occur Check, Algorithm 2.9.7, with “cut”, fail, and not, is the most effective and most widespread logic programming language. The fact that there is no theoretical guarantee regarding the correctness of its programs does not mean that its programs give the wrong result. It only means that the programmer has to be more careful with the basic terms, “cut” and not. The use of real PROLOG offers in every PROLOG program greater speed and effectiveness at the cost of the lack of a general completeness theorem.

A basic difference between Predicate Logic and PROLOG, both pure and real, is in the manipulation of a set of data S . PROLOG organizes the data, numbering the facts and the rules according to the order in which they are expressed. Furthermore, the order of predicates in the body of the rules is exactly the order in which they are written in the program. Therefore, rule:

$$A :- B_1, B_2. \quad (*)$$

does not match with rule

$$A :- B_2, B_1. \quad (**)$$

Although in Predicate Logic the two rules are equivalent because of the commutativity of \vee , in PROLOG they are considered different. Moreover, the choice between form (*) and form (**) directly affects the speed of execution of the program.

This procedural fashion in which PROLOG deals with data, causes problems in the program when there are facts of the form:

$$A :- A. \quad (1)$$

or more generally:

$$A :- B_1, \dots, A, \dots, B_k. \quad (2)$$

In other words, Horn clauses of the form:

$$A \rightarrow A$$

or

$$(B_1 \wedge \dots \wedge B_k \wedge A) \rightarrow A$$

which are logically correct (because they are equivalent to $\neg A \vee A$). Hence, whereas (1) and (2) do not give any information to the program, since they are clauses which are always true, any query of the form “? A .” causes an infinite loop through the recursive procedure. The program is unable to leave this loop by itself. Most of the infinite loops in PROLOG programs are due to the existence of trivial data of the form (1) or (2).

On the contrary, logically true Horn clauses of the form:

$$A :- \text{fail.}$$

or equivalently, in Predicate Logic, expressions of the form:

$$P \wedge \neg P \rightarrow A$$

where P can be any sentence of Predicate Logic and A is a Horn clause, are used in real PROLOG to express negation. The negation in real PROLOG is much stronger than the negation in Predicate Logic. In Predicate Logic, the negation of a clause A is determined from the logically true clause:

$$\neg A \leftrightarrow (A \rightarrow B \wedge \neg B)$$

or equivalently:

$$\neg A \leftrightarrow (A \rightarrow \square)$$

Hence, if

$$S \vdash_R A$$

then $\neg A$ and $\text{not}(A)$ of real PROLOG are unsatisfiable sentences, Definition 2.5.15 and subsection 3.6.2. If however

$$S \not\vdash_R A$$

i.e., if A is not provable from S with resolution, then, whereas in Predicate Logic we do not know if $S \vdash_R \neg A$, in real PROLOG, because of the Closed World Assumption (subsections 3.6.2 and 3.6.3), we will conclude $\text{not}(A)$.

Regardless of whether real PROLOG is complete or not, it is widely used and is the richest and most effective logic programming language. Definitely, controlling the correctness of the program always has to be carried out by the programmer and the user.

3.10 Exercises

3.10.1

Write in PROLOG the following expressions of everyday speech.

- (a) Chris hates studies.
- (b) Somebody loves George.
- (c) George is a friend of whomever he loves, if that person loves him too.
- (d) Mary likes whoever admires her.
- (e) Nick is afraid of whoever is bigger than him.
- (f) Nick is jealous of whomever Mary likes, if this person is not shorter than Nick.

Solution:

- (a) `hate(chris, studies).`
- (b) `love(X, nick).`
- (c) `is_friend(george, X) :- love(X, george), love(george, X).`
- (d) `likes(mary, X) :- admires(X, mary).`
- (e) `is_afraid(nick, X) :- bigger(X, nick).`
- (f) `is_jealous(nick, X) :- likes(mary, X), not(shorter(X, nick)).`

3.10.2

Write the PROLOG program corresponding to the proof of Exercise 2.12.37.

Solution:

```

t(a, b, c, d).                /* abcd trapez. with vertices a, b, c, d */
p(a, b, c, d) :- t(a, b, c, d).    /* ab || cd if abcd trapez. */
e(a, b, d, c; d, b) :- p(a, b, c, d).    /*  $\widehat{abd} = \widehat{cdb}$  if  $ab \parallel cd$  */
? e(a, b, d, c, d, b).            /*  $\widehat{abd} = \widehat{cdb}$  */

```

3.10.3

Write a PROLOG program with the countries of the EU and their respective capital cities as the database. Form the adequate PROLOG queries determining the capital city of a given country as well as the country corresponding to a capital city.

3.10.4

Define, for the following database, the relations:

- | | |
|------------------------------|-------------------------|
| (a) taller, taller(X, Y) | (b) tall, tall(X) |
| (c) normal, normal(X) | (d) short, short(X) |

Consider that a man, m , and a woman, f , are tall when they are respectively taller than 1.80 and 1.75, the corresponding limits for the relation short being 1.70 and 1.60.

```
height(bill,m,1.80).
height(nick,m,1.54).
height(jim,m,1.87).
height(paul,m,1.75).
height(john,m,1.66).
height(alex,m,1.76).
height(mary,f,1.73).
height(bill,f,1.61).
height(joan,f,1.68).
height(catherine,f,1.78).
```

Solution:

```
taller(X,Y):- height(X,_,Z), height(Y,_,W), Z > W.
tall(X):- height(X,m,Y), Y => 1.80.
tall(X):- height(X,f,Y), Y => 1.75.
normal(X):- height(X,m,Y), Y < 1.70, Y < 1.80.
normal(X):- height(X,f,Y), Y > 1.60, Y < 1.75.
short(X):- height(X,m,Y), Y =< 1.70.
short(X):- height(X,f,Y), Y =< 1.60.
```


3.10.5

Consider the following tables of materials and their suppliers for a warehouse containing spare parts of cars:

SUPPLIER	SUP.CODE	NAME	PROFESSION	CITY
	001	John	Manufacturer	ATHENS
	002	Nick	Importer	PATRAS
	010	John	Entrepreneur	SAL/ICA
	110	Nick	Importer	PIRAEUS

ARTICLE	ART.CODE	PRODUCT	MODEL	WEIGHT
	003	Oil	30	ATHENS
	004	Tyres	157/75	PATRAS
	005	Lamps	RAAI	SAL/ICA
	013	Oil	60	PIRAEUS

SUPPLIES	SUP.CODE	ART.CODE	QUANTITY
	001	005	150
	002	003	200
	010	004	030
	110	013	250

- (a) Represent the data in the above tables in a PROLOG database containing 2-ary predicates only.
- (b) Give the adequate Horn clauses for the queries:
 - (1) What are the names of the oil suppliers?
 - (2) Which city are tyre suppliers in, and which city are the lamp suppliers in?
 - (3) What does John supply?
 - (4) Which oil suppliers supply less than 400 tons and are established in Patras?
 - (5) Who are the lamp suppliers, and who are the tyre suppliers?

Solution:

(a) We introduce the predicates:

```

name(X,Y)           /* X supplier code, Y supplier name */
profession(X,Y)      /* X supplier code, Y supplier profession */
city(X,Y)           /* X supplier code, Y supplier city */
product(X,Y)         /* X product code, Y product name */
type(X,Y)           /* X product code, Y product type */
weight(X,Y)          /* X product code, Y product weight */
supp_article(X,Y)    /* X supplier code, Y article code */
quantity(X,Y)        /* X supplier code, Y product quantity */

```

We thus form the following database:

```

name(001,john).
name(002,nick).
name(010,john).
name(110,nick).
profession(001,manufacturer).
profession(002,importer).
profession(010,entrepreneur).
profession(110,importer).
city(001,athens).
city(002,patras).
city(010,salonica).
city(110,piraeus).
product(003,oil).
product(004,tyres).
product(005,lamps).
product(013,oil).
type(003,'30').
type(004,'157/75').
type(005,'RAAI').
type(013,'60').
weight(003,300).
weight(004,2000).
weight(005,10).
weight(013,500).
supp_article(001,005).
supp_article(002,003).
supp_article(010,004).
supp_article(110,013).

```

- (b) (1) `supp_oil(NameSupplier):-`
 `product(CodProduct,oil),`
 `supp_article(CodSupplier,CodProduct),`
 `name(CodSupplier,NameSupplier).`
- (2) `city_supp_tyres(CitySupplier):-`
 `product(CodProduct,tyres),`
 `supp_article(CodSupplier,CodProduct),`
 `city(CodSupplier,CitySupplier).`
- `city_supp_lamps(CitySupplier):-`
 `product(CodProduct,lamps),`
 `supp_article(CodSupplier,CodProduct),`
 `city(CodSupplier,CitySupplier).`
- (3) `supplies(NameSupplier,NameProduct):-`
 `name(CodSupplier,NameSupplier),`
 `supp_article(CodSupplier,CodProduct),`
 `product(CodProduct,NameProduct).`
 `? supplies(john,NameProduct).`
- (4) `supp_oil_less_400_patras(NameSupplier):-`
 `product(CodProduct,Oil),`
 `weight(CodProduct,WeightProduct),`
 `WeightProduct < 400,`
 `supp_article(CodSupplier,CodProduct),`
 `city(CodSupplier,patras),`
 `name(CodSupplier,patras).`
- (5) `supplies_lamps(NameSupplier):-`
 `product(CodProduct,lamps),`
 `supp_article(CodSuppliers,CodProduct),`
 `name(CodProduct,NameSupplier).`
- `supplies_tyres(NameSupplier):- product(CodProduct,tyres),`
 `supp_article(CodSupplier,CodProduct),`
 `name(CodProduct,NameSupplier).`

3.10.6

Analyse the PROLOG functioning of the following program, and give the corresponding state tree structure.

$A(a, b).$	$/ * 1 * /$
$B(X) :- C(X, Y).$	$/ * 2 * /$
$C(b, a) :- A(a, b).$	$/ * 3 * /$
$C(X, Y) :- A(X, Y).$	$/ * 4 * /$
$? B(X).$	

Solution:

The goal $B(X)$ is unified with the head of rule $/ * 2 * /$. The new subgoal is $C(X, Y)$. $C(X, Y)$ is unified with the head of $/ * 3 * /$ for X/b , Y/a . However, $A(a, b)$ is unified with fact $/ * 1 * /$, hence the program succeeds for X/b , Y/a , and PROLOG answers:

$$X = b$$

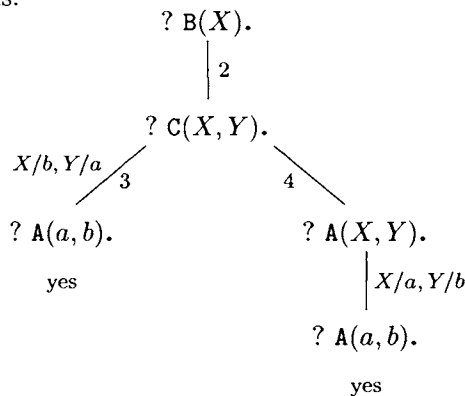
$$Y = a$$

PROLOG now frees variables X and Y from their values, and by backtracking it tries to satisfy the subgoal $C(X, Y)$ differently. $C(X, Y)$ is unified with the head of rule $/ * 4 * /$. The new subgoal is $A(X, Y)$, which is unified with fact $/ * 1 * /$ for X/a , Y/b . After having freed the variables and backtracking once again, PROLOG is no longer able to satisfy the goal or any subgoal, and answers:

$$X = a$$

$$Y = b$$

The state space tree is:



3.10.7

The police have found an unfortunate woman by the name of Suzanne murdered, with her head smashed by a blunt instrument. The main roles in this sad story are played by Alan, 35, a butcher as well as a thief, and John, 25, a soccer player who is sentimentally attached to both Suzanne and Barbara. Barbara is a 22 year old hairdresser who is married to Bert, a 50 year old lame joiner. During the investigation, a revolver was found in John's house. For the police, jealousy and a planned robbery are possible motives. Help them find the murderer.

Solution:

```

/* assumptions and claims formulae from the police investigation */
person(john,25,m,football_player).      /* m: male */
person(allan,35,m,butcher).
person(barbara,22,f,hairdresser).
person(bert,50,m,carpenter).
person(allan,35,m,pickpocket).
had_affair(barbara,john).
had_affair(barbara,bert).
had_affair(susan,john).
killed_with(susan,club).
motive(money).
motive(jealousy).
owns(bert,wooden_leg).
owns(john,pistol).

/* "common logic" assumptions */
operates_identically(wooden_leg,club).
operates_identically(bar,club).
operates_identically(pair_of_scissors,knife).
operates_identically(football_boot,club).
owns_probably(X,football_boot):- person(X,_,_,football_player).
owns_probably(X,pair_of_scissors):- person(X,_,_,_).
owns_probably(X,object):- owns(X,object).
```

```

/* assumptions about the murderer's motives - two categories of suspects: */
/* suspects by their capability of committing the murder, and */
/* suspects by their motives */
suspect_by_capability(X) :- killed_with(susan, Weapon),
                           operates_identically(Object, Weapon),
                           owns_probably(X, Object).

suspect_by_motive(X) :- motive(jealousy),
                        person(X, _ , m, _ ),
                        had_affair(susan, X).

suspect_by_motive(X) :- motive(jealousy),
                        person(X, _ , f, _ ),
                        had_affair(X, Man),
                        had_affair(susan, Man).

suspect_by_motive(X) :- motive(money),
                        person(X, _ , _ , pickpocket).

/* non-probabilistic "common logic" assumption */
mostly_suspected(X) :- suspect_by_capability(X),
                       suspect_by_motive(X).

```

By its definition, the predicate `mostly_suspected` has no probabilistic interpretation, and depends only on the two predicates `suspect_by_capability` and `suspect_by_motive`.

Justify by a full track down the answers to the following PROLOG queries:

```
? suspect_by_capability(X).
```

```
X = bert
```

```
X = john
```

```
? suspect_by_motive(X).
```

```
X = john
```

```
X = barbara
```

```
X = allan
```

```
? mostly_suspected(X).
```

```
X = john
```

3.10.8

Reading and evaluating kilometric distances on a map often proves to be a very complex procedure in the preparation of an excursion. Hence a PROLOG program would be very useful.

Solution:

As a sample program, consider:

```

/* data from a map */
road(kalamata,tripoli,90).
road(tripoli,argos,60).
road(argos,korinthos,49).
road(korinthos,athens,83).

/* rules for the calculation of kilometric distances */
route(Town1,Town2,Distance):- road(Town1,Town2,Distance).
route(Town1,Town2,Distance):- road(Town1,X,Dist1),
                                route(X,Town2,Dist2),
                                Distance = Dist1 + Dist2.

```

What is the distance between Kalamata and Korinthos?

$$\left[\begin{array}{l} ? \text{ route(kalamata,korinthos,X)}. \\ X = 199 \end{array} \right]$$
3.10.9

Give the recursive definition of the predicate `factorial(X,Y)`, Y being $X!$, based on:

$$(a) \quad 1! = 1$$

$$(b) \quad n! = (n-1)! * n$$

Construct the state space tree of the program for the query:

```
? factorial(3,X).
```

Solution:

Here (a) gives the bound condition and (b) gives the recursive step. The definition we are seeking is thus:

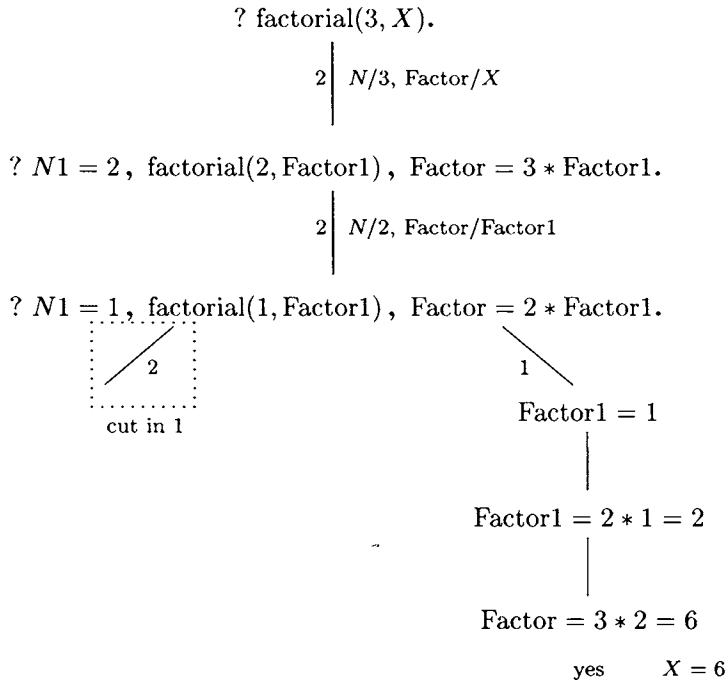
```
factorial(1,1):- !.                                /* 1 */
factorial(N,Factor):- N1 = N - 1,
                                factorial(N1,Factor1),
                                Factor = N * Factor1.    /* 2 */
```

The cut in formula /* 1 */ has been used in order to prevent PROLOG from backtracking during the calculation of 1!.

The state space for the query:

```
? factorial(3,X).
```

is depicted by the following tree:



3.10.10

Define recursively the predicate $\text{exp}(x, y, z)$, meaning “ $x^y = z$ ”, which defines the exponential function based on:

$$\begin{aligned}x^0 &= 1 \\ x^y &= x \cdot x^{y-1}\end{aligned}$$

3.10.11

Assume we wish to define a predicate declaring the number of parents of a person x . According to the Bible, we have to declare that Adam and Eve had no parents. We thus write the program $P1$:

```
/* program P1 */
number_parents(adam,0):-!.
number_parents(eve,0):-!.
number_parents(X,2).
/* 1 */
/* 2 */
/* 3 */
```

where we use the cut to avoid backtrackings in queries of the form:

```
? number_parents(adam, X).
```

and

```
? number_parents(adam, 0).
```

- (a) What will PROLOG answer to the query:

```
? number_parents(eve, 2). (*)
```

and why?

- (b) What will PROLOG answer to (*) if, instead of the program $P1$, we use the following program $P2$:

```
/* program P2 */
number_parents(adam,N):-!, N=0.
number_parents(eve,N):-!, N=0.
number_parents(X,2).
/* 1 */
/* 2 */
/* 3 */
```

and what is the answer to

```
? number_parents(X,Y).
```

given by the program *P2*?

3.10.12

Define with a PROLOG program the absolute value of an integer.

3.10.13

Write a PROLOG program checking whether an element (a person or a list) is part of a list *L*, and examining all the elements of *L*.

Solution:

```
member(H,[H : _]).
member(I,[ _ : T]) :- member(I,T).
```

3.10.14

Check whether the list *H* is member of the list *L*, and find the first element of *L*. Apply this to the list *L* = [a, b, c].

Solution:

```
member(H:[H _]) :- !.                               /* 1 */
member(I,[ _ : T]) :- member(I,T).                  /* 2 */
```

The only solution found by PROLOG for the query:

```
? member(X,[a,b,c]).
```

is $X = a$. The existence of the cut, !, in clause /* 1 */ makes the finding of other possible solutions impossible.

3.10.15

Write a program collecting the first *n* elements of a list.

3.10.16

Write a program defining the subtraction of two lists.

Solution:

```
subtract(L,[ ],L):- !.
subtract([H:T],L,U):- member(H,L), !, subtract(T,L,U).
subtract([H:T],L,U):- !, subtract(T,L,U).
subtract( _ , _ ,[ ]).
member(H,[H: _ ]).
member(I,[ _ :T]):- member(I,T).
```

3.10.17

Define by means of a program, the relations

member, subset and intersection

of set theory.

Solution:

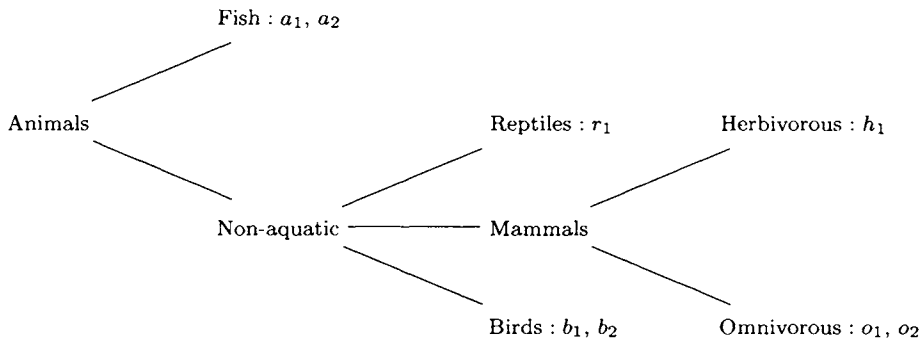
```
/* ∈ */
member(H,[H: _ ])
member(I,[ _ :T]):- member(I,T).

/* ⊆ */
subset([H:T],I):- member(H,I), subset(T,I).
subset([ ],I).

/* ∩ */
intersection([ ],X,[ ]).
intersection([X:T],I,[X:Z]):- member(X,I), !, intersection(T,I,Z).
```

3.10.18

Assume we are given a list of eight animals, $V = [a1, a2, r1, b1, b2, h1, o1, o2]$, which are classified as follows:



Write a PROLOG database describing the above classification, and a PROLOG program replying to the queries:

- What is the classification of the animal X ?
- Which animals have classification Y ?

Solution:

```

aquatic(aquatic,[a1,a2]).
reptile(reptile,[r1]).
bird(bird,[b1,b2]).
mammal(V,X):- herbivorous(V,X); omnivorous(V,X).
herbivorous(herbivorous,[h1]).
omnivorous(omnivorous,[o1,o2]).
classification(A,C):- animal(C,L), member(A,L).
animal(V,X):- aquatic(V,X); terrestrial(V,X).
terrestrial(terrestrial(V),X):- reptile(V,X); bird(V,X); mammal(V,X).
  
```

In answer to the classification queries:

- ? classification(X,C).
- ? animal(X,Y).

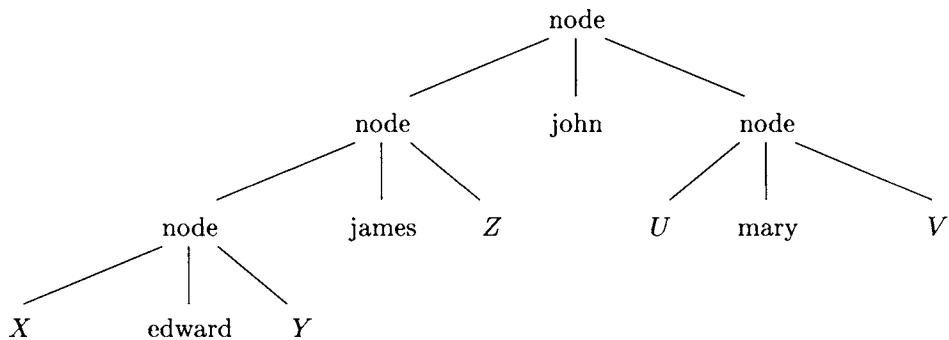
3.10.19

Write a PROLOG program displaying a list of names in alphabetical order. The names in the list must be separated by a full stop, and the word “stop” must appear after the last name.

Solution:

The alphabetic ordering of the names can be achieved by means of a tree, every node of which is the predicate `node(L,W,R)`, interpreted as “*L* is the next left node of the present node, *R* is its next right node, and the next middle node is the name just read by the program”.

Each name will be compared to the origin, and recursively to the next left and right nodes of the corresponding node. This procedure will end when the next node is a variable, or when the name is already in the tree. For example, for the names `john.mary.james.edward.stop.` we will have the following tree:



The program we are seeking is:

```

order(X):- reading(X), name(W,W1), classify(W1,X,Y), order(Y).
order(X):- nl, write('ordered words:'), nl,
            write_tree(X), nl, nl, write('Done.'), nl.
reading(W):- read(W), W == stop, !, fail.
classify(W,node(L,W,R),node(L,W,R)):- !.
classify(W,node(L,W1,R),node(U,W1,R)):-
    lower(W,W1), !, classify(W,L,U).
classify(W,node(L,W1,R),node(L,W1,U)):- !, classify(W,R,U).
lower([ ], _ ):- !, lower(Y,T).
lower([W : Y],[Z,T]):- X < Z.
write_tree(X):- var(X), !.
write_tree(node(L,W,R)):- !, write_tree(L), name(W,W1),
    write(W,W1), write(' '), write_tree(R).
  
```

To run the program, the following instruction must be given:

```
:- order(X).
```

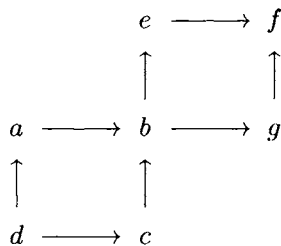
and then the list of names to be ordered.

3.10.20

Write a PROLOG program forming a tree with a given list of numbers and printing that list as a tree with its elements ordered.

3.10.21

Write a program representing the diagram:



and finding the next node of c and b .

Solution:

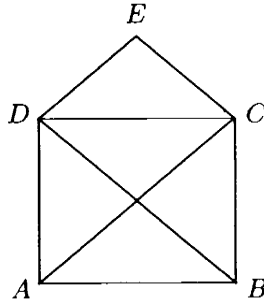
The arrangement of the nodes is declared by the predicate `arc(x,y)`, which is interpreted as “an edge starting at x and ending at y ”. The program we are seeking is:

```

arc(a,b).
arc(c,b).
arc(d,a).
arc(d,c).
arc(b,e).
arc(e,f).
arc(f,g).
arc(b,g).
? arc(c,x).
? arc(g,y).
```

3.10.22

Write a program drawing the following figure with a continuous line:



Solution:

```

draw(G, [P, Q : L]) :- choose(G, link(P, Q), G1), draw(G1, [Q, L]).
draw([ ], [Q]).
choose([link(P, Q) : G], link(P, Q), G).
choose([link(P, Q) : G], link(Q, P), G).
choose([E : G], F, [E, G1]) :- choose(G, F, G1).
: draw([link(a, b), link(a, c), link(a, d), link(b, c),
        link(b, d), link(c, d), link(c, e), link(d, e)], L), write(L).

```

3.10.23

Write a program describing given parts of the plane, and determining whether:

- (a) a given point (a, b) lies inside a circle with center (c, d) and radius r .
- (b) three points of the plane with coordinates

$$(x_1, y_1), \quad (x_2, y_2) \quad \text{and} \quad (x_3, y_3)$$

respectively are collinear.

Solution:

The given points are described by the predicate:

`point(x, y)`

where x and y are the coordinates of the points.

The equation of the interior of the circle is:

$$(x - c)^2 + (y - d)^2 < r^2$$

In the Cartesian plane, the three points are collinear if:

$$\frac{y_3 - y_1}{x_3 - x_1} = \frac{y_2 - y_1}{x_2 - x_1}$$

Finally, the predicate:

`inside(point(x, y), (c, d), r)`

declares that:

`point(x, y)`

lies inside the circle with centre (c, d) and radius r . The necessary built-in predicates and the formalism of arithmetical operations can be found in subsection 3.5.4.

3.10.24

Define combinations of the elements of a given list.

3.10.25

Write an algorithm checking whether a given point coincides with one of the nodes of a given tree, every node of which has at most two next nodes (Definition 2.8.9). In the case that the given point does not belong to the nodes of the tree, the algorithm must concatenate that point as a new node of trees.

3.10.26

George, Tim, John and Bill are soccer fans, and Nick and Jim are supporters of teams A and B respectively. Nick likes whomever supports team A , whereas George likes whomever is a soccer fan and does not support team B . Formulate the above claims in a PROLOG database, and answer the following queries:

- (a) Whom does Nick like?
- (b) Does George like Bill?
- (c) Whom does George like?

Solution:

```

/* data claims */
support(nick, A).
support(jim, B).
is_occupied(george, football).
is_occupied(tim, football).
is_occupied(john, football).
is_occupied(bill, football).
is_occupied(X, football) :- support(X, B).
is_occupied(X, football) :- support(X, A).
love(nick, X) :- support(X, A).
love(george, X) :- is_occupied(X, football), not(support(X, B)).

/* queries and answers */
? love(nick, X).
X = jim
? likes(george, bill).
yes
? likes(george, X).
X = george
X = tim
X = john
X = bill

```

3.10.27

Assume we are given a PROLOG program Q . Examine the queries:

- $$\begin{aligned}
 (1) \quad & ? \text{not}(p). \\
 (2) \quad & ? \text{not}(\text{not}(p)). \\
 (3) \quad & ? \text{not}(\text{not}(\text{not}(p))). \\
 & \quad \cdot \quad \cdot \quad \cdot \\
 (2n) \quad & ? \underbrace{\text{not}(\dots(\text{not}(p))\dots)}_{2n}. \\
 (2n+1) \quad & ? \underbrace{\text{not}(\dots(\text{not}(p))\dots)}_{2n+1}.
 \end{aligned}$$

where p is a PrL predicate without free variables.

Solution:

Assume p succeeds in program Q . Then the goal $\text{not}(p)$ fails, and PROLOG will answer “no” in query (1), “yes” in query (2), “no” in query (3), \dots , “yes” in query $(2n)$, and “no” in query $(2n + 1)$, by the definition of not .

If p fails, then $\text{not}(p)$ succeeds, and PROLOG will reply “yes” in query (1), “no” in query (2), “yes” in query (3), \dots , “no” in query $(2n)$, and “yes” in query $(2n + 1)$.

In general, the predicate $\text{not}^{2n}(p)$ takes the same value as p , while $\text{not}^{2n+1}(p)$ takes the same value as $\text{not}(p)$, exactly like the PL negation.

The differences between not and PL negation were examined in sections 3.6.1 and 3.9.

3.10.28

Write a simple parser for the analysis of the sentences:

The child sleeps and

The child eats apples

Solution:

```

sent(X,Y):- np(X,U), vp(U,Y);
np(X,Y):- det(X,U), noun(U,Y).
vp(X,Y):- iverb(X,Y).
vp(X,Y):- tverb(X,U), np(U,Y).
det([the : Y],Y).
noun([child : Y],Y).
noun([apples : Y],Y).
iverb([sleeps : Y],Y).
tverb([sleeps : Y],Y).
tverb([eats : Y],Y).

```

3.10.29

Form a simple parser analysing sentences like:

“Mary ate the cake”

3.10.30

The following sentence of the English language is given:

P : George is out shopping or he is at home.

Give the corresponding clause for P as well as the conclusions which can be inferred, given a database consisting of this sentence.

Solution:

Let A and B be the sentences

A : George is out shopping.

B : George is at home.

In the context of PrL, P takes the form:

$$P : A \vee B$$

which has two logically equivalent sentences:

$$\neg A \rightarrow B$$

$$\neg B \rightarrow A$$

The clauses of those sentences are:

$$B :- \text{not}(A). \quad \quad \quad /* \ 1 \ */$$

$$A :- \text{not}(B). \quad \quad \quad /* \ 2 \ */$$

If the database of a PROLOG Program consists of clauses $/* \ 1 \ */$ and $/* \ 2 \ */$, then the two queries:

? A.

? B.

will not be answered, since the control of the program's execution will end in a loop, as one can easily see by analysing the corresponding state space.

One way to arrive at a solution is to form two different programs, consisting of $/* \ 1 \ */$ and $/* \ 2 \ */$ respectively. In that case, the corresponding state spaces of the programs for the same queries will be:

PROGRAM I

$$B :- \text{not}(A). \quad \quad \quad /* \ 1 \ */$$

? B.

? not(A).

PROGRAM II

$$A :- \text{not}(B). \quad \quad \quad /* \ 2 \ */$$

? A.

? not(B).

PROLOG answers both queries of each of the two programs with "yes" by means of the NF rule. Thus, in PROGRAM I, "not(A)" and "B" are inferred; whereas in PROGRAM II, "A" and "not(B)" are inferred. In other words, we observe that for the sentence P , there are two different, and contradictory, sets of conclusions. Problems of this kind occur when we have sentences such as P , which cannot take a Horn clause form, and for which we cannot determine whether one, or more than one of the constituent sentences is true or false. For example, we cannot determine the truth value either of A or B . As we have already seen in Definition 1.9.7, a Horn clause must contain at least one negative literal, and that is not the case in $A \vee B$, where A and B are sentences without negation.

Bibliography

1. Suggestions for Further Reading

1.1 *Propositional Logic*

For the history and evolution of the subject, read [Heij67, Boch62] and [NeSh93], which contains an extensive bibliography.

For axiomatization and rules of inference, read [Schm60, Chur56, HiAc28, Hami78, Raut79, Mend64]. For an approach at a more advanced level, see the propositional part of [Klee52].

For the tableaux method, see [Smul68, NeSh93].

For Boolean and other algebras for logic, read [RaSi70, Rasi74].

For modal logic, read [Chel80, HuCr68, NeSh93, Raut79, Schm60]. For an advanced study of the relation of modal logic to predicate logic, read [Bent83].

For intuitionistic logic, see [Brou75, Dumm77, Fitt69, NeSh93, Raut79].

For resolution, see the propositional part of [ChLe73, Dela87].

1.2 *Predicate Logic*

To see the beginning and evolution of the subject, see [Heij67, Boye68, Boch62] and [NeSh93], which contains an extensive bibliography.

For the axiomatic method, read [Chur56, Hami78, Mend64, HiAc28], and the more advanced [Klee52].

For the tableaux method, see [Smul68, NeSh93].

For Herbrand's theorem, see [Herb30] in [Heij67], and also [ChLe73, Dela87, NeSh93, NiMa95].

For decidability, read [Acke54, Chur56, Klee52].

For resolution, [ChLe73, Dela87, NiMa95, NeSh93] and the standard [Lloy87].

1.3 Logic Programming

For logic programming in general, see [ChLe73, Dela87, Kowa79, Kowa90, Lloy87, NeSh93, NiMa95].

For PROLOG, read [Brat90, ClMc84, ClMe94, Dela87, NiMa95, StSh86].

For the treatment of negation, read [Kowa90, Lloy87, NeSh93, NiMa95, Shep88, Shep92].

For metaprogramming, see [Kowa90] and [HiLl94], which develops the new programming language GÖDEL.

For nonmonotonic logics, see [MaTr93].

2. References

- [Acke54] Ackermann, W., *Solvable Cases of the Decision Problem*, North-Holland Pub. Co., 1954.
- [Bent83] van Benthem, J. F. A. K., *Modal Logic and Classical Logic*, Bibliopolis, Napoli, 1983.
- [Besn89] Besnard, P., *An Introduction to Default Logic*, Springer-Verlag, 1989.
- [Beth68] Beth, E. W., *The Foundations of Mathematics; a Study in the Philosophy of Science*, 3rd edn., North-Holland Pub. Co., 1968.
- [Boch62] Bochenski, J. M., *Formale Logik*, (in German), 2nd edn., Verlag Karl Alber, Freiburg-Muenchen, 1962. Translated as: *A History of Formal Logic*, (in English), Thomas, I., tr., Chelsea Pub. Co., 1970.
- [Boye68] Boyer, C. B., *A History of Mathematics*, Princeton University Press, 1985. (Paperback version of the 1968 Wiley edition.)
- [Brat90] Bratko, I., *PROLOG Programming for Artificial Intelligence*, 2nd edn., Addison-Wesley Pub. Co., 1990.
- [Brou75] Brouwer, L. E. J., *Collected Works*, Heyting, A., ed., North-Holland Pub. Co., 1975.

- [BuSh84] Buchanan, B. G., Shortliffe, E. H., eds., *Rule-based Expert Systems: the MYSIN experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Pub. Co., 1984.
- [CCPe85] Coelho, H., Cotta, J. C., Pereira, L. M., *How to Solve it with PROLOG*, 4th edn., Ministério do Equipamento Social, Laboratório Nacional de Engenharia Civil, Lisbon, 1985.
- [Chel80] Chellas, B. F., *Modal Logic, an Introduction*, Cambridge University Press, 1980.
The text most referred to in computer science literature.
- [ChLe73] Chang, C-L., Lee, C-T., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.
- [Chur36] Church, A., *A Note on the Entscheidungsproblem*, Journal of Symbolic Logic, **1**, 1936, pp. 40–41, and correction, *ibid.*, pp. 101–102.
- [Chur56] Church, A., *Introduction to Mathematical Logic, Volume I*, Princeton University Press, 1956.
- [Clar78] Clark, K. L., *Negation as Failure*, in [GaMi78], pp. 293–322.
- [ClMc84] Clark, K. L., McCabe, F. G., *MICRO-PROLOG: Programming in Logic*, Prentice-Hall International, 1984.
- [ClMe94] Clocksin, W. F., Mellish, C. S., *Programming in PROLOG*, 4th edn., Springer-Verlag, 1994.
- [Cohe90] Cohen, J., *Constraint Logic Programming Languages*, Communications of the ACM, **33**, 7, July 1990, pp. 52–68.
- [Colm87] Colmerauer, A., *An Introduction to PROLOG III*, in [Espr87], Part I, North-Holland, 1987, pp. 611–629.
- [Colm90] Colmerauer, A., *An Introduction to PROLOG III*, in [Lloy90], pp. 37–79.
- [Col90a] Colmerauer, A.: *An Introduction to PROLOG III*, Communications of the ACM, **33**, 7, July 1990, pp. 69–90.
- [Curr63] Curry, H. B., *Foundations of Mathematical Logic*, McGraw-Hill, 1963. Dover Pub., 1977.

- [Dela87] Delahaye, J.-P., *Formal Methods in Artificial Intelligence*, Howlett, J., tr., North Oxford Academic Publishers Ltd., London, 1987. Wiley, New York, 1987.
- [DiSc90] Dijkstra, E. W., Scholten, C. S., *Predicate Calculus and Program Semantics*, Springer-Verlag, 1990.
- [Dumm77] Dummett, M. A. E., *Elements of Intuitionism*, Clarendon Press, Oxford, 1977.
- [Espr87] ESPRIT '87: *Achievements and Impacts*, Proceedings of the 4th Annual ESPRIT Conference, Brussels, September 28–29, 1987.
- [Fitt69] Fitting, M. C., *Intuitionistic Logic, Model Theory and Forcing*, North-Holland Pub. Co., 1969.
- [Fitt90] Fitting, M. C., *First-Order Logic and Automated Theorem Proving*, Springer-Verlag, 1990.
- [GaMi78] Gallaire, H., Minker, J., eds., *Logic and Data Bases*, Proceedings of the Symposium on Logic and Data Bases held at the Centre d'Études et de Recherches de L'École Nationale Supérieure de L'Aéronautique et de L'Espace de Toulouse (C.E.R.T.), Toulouse, November 16–18, 1977, Plenum Press, New York, 1978.
- [Hami78] Hamilton, A. G., *Logic for Mathematicians*, Cambridge University Press, 1978.
There is a rev. edn., Cambridge University Press, 1988
- [Heij67] van Heijenoort, J., *From Frege to Gödel. A Source Book in Mathematical Logic, 1879–1931*, Harvard University Press, 1967.
It contains the most important papers in logic from 1879 to 1931.
- [Herb30] Herbrand, J., *Recherches sur la théorie de la démonstration*, thesis at the University of Paris. Chapter 5 translated as: *Investigations in Proof Theory: the Properties of True Propositions*, Dreben, B., van Heijenoort, J., tr., in [Heij67], pp. 525–581.
- [HiAc28] Hilbert, D., Ackermann, W., *Grundzüge der theoretischen Logik*, Springer-Verlag, Berlin, 1928. The 2nd edn. translated as: *Principles of Mathematical Logic*, Hammond, L. M., Leckie, G. G., Steinhardt, F., tr., Luce, R. E., ed., Chelsea Pub. Co., 1950.

- [HiLl94] Hill, P., Lloyd, J. W., *The GÖDEL Programming Language*, MIT Press, 1994.
- [HuCr68] Hughes, G. E., Cresswell, M. J., *An Introduction to Modal Logic*, Methuen and Co. Ltd, 1968. Routledge, 1989.
- [Klee52] Kleene, S. C., *Introduction to Metamathematics*, corrected reprint of the 1952 edn., North-Holland Pub. Co., 1971.
This book contains unexcelled explanations of Gödel's theorems.
- [Kowa79] Kowalski, R. A., *Logic for Problem Solving*, North-Holland Pub. Co., 1979, reprinted, 1986.
- [Kowa90] Kowalski, R. A., *Problems and Promises of Computational Logic*, in [Lloy90], pp. 1-36.
- [Lloy87] Lloyd, J. W., *Foundations of Logic Programming*, 2nd edn., Springer-Verlag, 1987.
This is the standard exposition of the subject.
- [Lloy90] Lloyd, J. W., ed., *Computational Logic: Symposium Proceedings, Brussels, November 13-14, 1990*, ESPRIT Basic Research Series, Springer-Verlag, 1990.
- [MaTr93] Marek, V. W., Truszczyński, M., *Nonmonotonic Logic: Context Dependent Reasoning*, Springer Verlag, 1993
- [Mend64] Mendelson, E., *Introduction to Mathematical Logic*, Van Nostrand Company, Inc., 1964.
There is a 3rd edn., Wadsworth & Brooks-Cole, 1987.
- [Meta85] Metakides, G. *Mathematical Logic*, (in Greek), University of Patras, Greece, 1985.
- [Meta92] Metakides, G. *From Logic to Logic Programming*, (in Greek), Kardamitsa Pub., Athens, Greece, 1992.
- [Mink88] Minker, J., ed., *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann Pub., Los Altos, CA, 1988.
- [Mitc86] Mitcie, D., ed., *Expert Systems in the Micro-Electronic Age*, Edinburgh University Press, 1979, reprinted 1986.

- [Mosc92] Moschovakis, V. N., ed., *Logic from Computer Science: Proceedings of a Workshop held November 13–17, 1989*, MSRI Publications 21, Springer-Verlag, 1992.
- [NeSh93] Nerode, A., Shore, R. A., *Logic for Applications*, Springer-Verlag, 1993.
- [Nils71] Nilsson, N. J., *Problem-solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.
- [NiMa95] Nilsson, U., Maluszynski, J., *Logic, Programming and PROLOG*, 2nd edn., John Wiley and Sons, 1995.
- [OlRu87] Olson, J. R., Rueter, H. H., *Extracting Expertise from Experts: Methods for Knowledge Aquisition*, Technical Report, University of Michigan, Cognitive Science and Machine Intelligence Laboratory, N^o 13, 1987.
- [Quin86] Quinlan, J. R., *Discovering Rules by Induction from Large Collections of Examples*, in [Mitc86], pp. 168–201
- [Rasi74] Rasiowa, H., *An Algebraic Approach to Non-Classical Logics*, North-Holland / American Elsevier Pub. Co., 1974.
In this book, numerous logic calculi are studied algebraically.
- [RaSi70] Rasiowa, H., Sikorski, R., *The Mathematics of Metamathematics*, 3rd edn., PWN-Polish Scientific Publishers, 1970.
The standard book for an algebraic approach to logic.
- [Raut79] Rautenberg, W., *Klassische und nichtklassische Aussagenlogik*, Vieweg, 1979.
- [Reit78] Reiter, R., *On Closed World Data Bases*, in *Logic and Databases*, Plenum Press, New York, 1978.
- [Robi65] Robinson, J. A., *A Machine-Oriented Logic Based on the Resolution Principle*. *Journal of the Association for Computing Machinery*, **12**, 1, 1965, pp. 23–41.
- [Schm60] Schmidt, H. A., *Mathematische Gesetze der Logik I*, Springer-Verlag, 1960.
A most comprehensive and elaborated text for propositional logic in the German literature.

- [Schw71] Schwabhäuser, W., *Modelltheorie, Bd. I*, Hochultaschenbuecher, Band 813, 1971.
- [Shap87] *Concurrent PROLOG: Collected Papers*, Shapiro, E., ed., vols. 1 and 2, MIT Press, 1987.
- [Shep88] Shepherdson, J., *Negation in Logic Programming*, in [Mink88], pp. 19–88.
- [Shep92] Shepherdson, J., *Logics for Negation as Failure*, in [Mosc92].
- [Smul68] Smullyan, R. M., *First Order Logic*, Springer-Verlag, 1968. Dover Publ., 1995.
- [StSh86] Sterling, L., Shapiro, E., *The Art of PROLOG: Advanced Programming Techniques*, MIT Press, 1986.
- [Thay88] Thayse, A., ed., *From Standard Logic to Logic Programming: Introducing a Logic Based Approach to Artificial Intelligence*, John Wiley & Sons, 1988.
- [Turi37] Turing, A. M., *On computable numbers with an application to the Entscheidungsproblem*, Proc. London Math. Soc., **42**, 1937, pp. 230–265. A correction, *ibid*, **43**, 1937, pp. 544–546.
- [Turn84] Turner, R., *Logics for Artificial Intelligence*, Ellis Horwood Series in Artificial Intelligence, Halsted Press, 1984.
- [Watt90] Watt, D. A., *Programming Language Concepts and Paradigms*, Prentice Hall, 1990.
- [WeKu84] Weiss, S. M., Kulikowski, C. A., *A Practical Guide to Designing Expert Systems*, Chapman and Hall Ltd., London, 1984.
- [WiBe89] Winston, P. H., Horn, B. K. P., *LISP*, 3rd edn., Addison-Wesley, Reading, MA, 1981.
- [Xant90] Xanthakis, S., *PROLOG, Programming Techniques*, (in Greek), New Technology Editions, Athens, 1990.
- [Zeno76] Diogenes Laertius, *Lives of Eminent Philosophers, Diogenes Laertius, Volume II*, Book VII, Zeno, para. 76, Hicks, R. D., tr., Loeb Classical Library, pp. 182, 184 (Greek), pp. 183, 185 (English), Heinemann, London, and Harvard University Press, 1925.

Index of Symbols

\wedge	2, 17, 32, 96, 137, 260	$+$	97, 256–259
\neg	2, 17, 32, 96, 137, 258, 260, 262	$*$	97, 256, 258–259
\vee	2, 17, 32, 96, 137, 260	ϵ	116
\rightarrow	3, 17, 32, 96, 137, 224, 260	$.$	206, 215–216, 229–231
\leftrightarrow	3, 17, 32, 96, 137, 260	$—$	217, 223–224, 249
$,$	6, 47, 96, 218	$?$	219–220
$($	6, 96	$[]$	229–231
$)$	6, 96	$/ * */$	236
\leftarrow	9	$!$	240
\sim	10–11, 264	$==$	255, 259
\sqcup	10–11	$==:$	255, 259
\sqcap	10–11	$\backslash ==$	256, 259
\rightsquigarrow	10–11	$= \backslash =$	256, 259
\longleftrightarrow	10–11	$-$	256, 259
$:=$	13	$/$	256, 259
\models	14, 120, 163	$>$	256, 259
$\not\models$	14	$<$	256, 259
\diamond	14	$=>$	256, 259
\equiv	15	$=<$	256, 259
\Leftrightarrow	21	\models_{CWA}	274
\Rightarrow	21	\models_{NM}	274
\emptyset	22	$\not\models_{\text{NM}}$	274
$ $	30	$\backslash =$	282
$:$	30, 231	\mathcal{A}	116
\vdash_B	36, 163	\mathcal{A}^*	121
\vdash	39, 104, 163	\mathcal{A}^f	28
\square	46–47	\mathcal{A}_H	134
$;$	47, 219	\mathcal{A}^t	28
$:-$	47–48, 218	AI	275
\vdash_R	52, 163	allowed_transition	238
\forall	96, 104, 137	append	250–251
\exists	96, 137	assert	217, 252
$=$	97, 108, 255, 259	asserta	253
\leq	97	assertz	253

- atom 257
- atomic 257
- CNF 29, 44, 125
- Con* 21, 125
- ConCWA* 274
- Compl(P) 267
- consult 253, 255
- CST 138, 141
- CWA 262
- div 256
- DNF 29, 125
- DS 156
- E* 100
- f* 10–11
- fail 271
- find_path 240
- free(x, t, σ) 103
- GHC 284
- gt 206
- GU 155
- Int* 24
- integer 257
- is 255–256, 259
- KES 278
- \mathcal{L} 114, 116
- \mathcal{L}^* 120
- \mathcal{L}_A 97
- member 249
- MGU 156
- mod 256
- NF 264
- nl 211
- no 212, 263
- not 272
- NM 274
- nonvar 257
- OC 157
- op 257
- PL 5
- PNF 125–126
- PrL 96
- $R(S)$ 50
- $R^n(S)$ 51
- $R^*(S)$ 51
- read 253–255
- real 211, 257
- retract 217, 252–253
- SNF 125, 129
- SolutionPath 239
- subform(A) 9
- t 10–11
- Taut* 22
- var 257
- write 211, 253–255
- yes 212

Index of Terms

- adequacy, of logical connectives 27
- adequate set 28
- agrees, interpretation, with
 - branch 164
- agrees, truth valuation, with
 - branch 56
- and 6
- algebra, truth values 10
- algorithm, construction of
 - a CNF 44–45
- algorithm, construction of
 - a CST 140–141
- algorithm, construction of
 - an Herbrand universe 132
- algorithm, construction of
 - a PNF 127–128
- algorithm, construction of
 - a semantic tree 145–146
- algorithm, construction of
 - an SNF 129
- algorithm, depth-first search 239–240
- algorithm, Herbrand 213
- algorithm, PL satisfiability of
 - a sentence 150–152
- algorithm, reduction to
 - clausal form 153–154
- algorithm, unification 154–155, 157–158, 232–235
- allowed transition 238
- alphabet, PROLOG 222
- alphabet, propositional logic 5–6
- alphabetic listing problem 300–303
- anonymous variables 223
- aquisition systems, knowledge 276
- arc, of a tree 144
- Aristotelian, syllogistic laws 20
- Aristotelian, world 14
- arithmetic, language of 97, 134
- arithmetic, PROLOG 256
- artificial intelligence 275
- arity, of a predicate 96
- ARITY-PROLOG 274
- associativity 44, 106
- associativity, operator 259
- atomic formula 97
- atomic propositions 6
- atomic tableau, predicate logic 136–137
- atomic tableau, propositional logic 32
- atomic term 224
- atoms 6, 97
- atoms, PROLOG 222, 224
- automatic theorem proving 31, 131
- axiomatic proofs 38
- axiomatic proofs,
 - compactness theorem 64
- axiomatic proofs,
 - completeness theorem 64, 167
- axiomatic proofs,
 - soundness theorem 64, 167
- axiomatic system, predicate logic 104
- axiomatic system, propositional logic 38, 42
- axioms, of equality 108
- axioms, selection of 42
- backtracking 209, 236–240, 242
- BASIC 209–210
- BASIC program 211
- Beth-deduction 60
- Beth-proof 36, 163
- Beth-provable 36, 42, 142, 163–164
- Beth-refutable 36, 142
- Beth-refutation 36
- Beth semantic tableau 31
- binding, of connectives 9
- binding, of variables 232
- body, of a clause 48
- body, of a list 229

- body, of a rule 216
- Boolean algebra 11
- Boolean valuation 10–11
- bound occurrences, of variables 99
- bound of a recursion 248
- bound variable 99
- boundary conditions, for a relation 247
- branch, complete, of a semantic tree 146
- branch, contradictory, of a semantic tableau 34
- branch, contradictory, of a semantic tree 145
- branch, tree 144
- Brouwer 18
- calculus, λ 214
- characters, special, PROLOG 222
- check, occur 157
- Church 131
- clause, empty 46
- clause, Horn 48, 111, 204–205, 215
- clause, predicate logic 110
- clause, propositional logic 46
- clause, unit 48
- clauses, set of 46, 131
- closed formula 100
- closed world assumption 213, 262
- COBOL 209
- comma 6, 47, 96, 218
- commands 209
- commands, imperative 210
- comments, PROLOG programs 236
- commutativity 44, 107
- compactness theorem,
 - axiomatic proofs 64
- compactness theorem, deductions 64
- compactness theorem,
 - tableaux proofs 165
- complete branch, of a tree 146
- complete definition, of a
 - predicate in a program 267
- complete induction 7
- complete systematic tableau 138, 141
- complete tableau 34
- completeness theorem, axiomatic
 - proofs, predicate logic 167
- completeness theorem, axiomatic
 - proofs, propositional logic 64
- completeness theorem, deductions 62
- completeness theorem,
 - resolution 65–66, 165
- completeness theorem,
 - semantic tableaux 59
- completeness theorem,
 - tableaux proofs 164
- completion of a program 267
- complex queries 220
- composition, of substitutions 101
- compound propositions 6, 20
- concatenate 250
- concurrent PROLOG 284
- conditions, boundary,
 - for a relation 247
- conditions, marginal,
 - for a relation 247
- conjunction 2, 6, 46
- conjunctive normal form, propositional
 - logic 29, 44
- conjunctive queries 220
- connectives, logical 6, 10, 96
- consequence, predicate logic 125
- consequence, propositional logic 21
- consistent, maximal set 77
- consistent, semantically 23
- consistent, sentence 122
- consistent, set of sentences 122
- constants, predicate logic 94, 97
- constants, PROLOG 222, 224
- constraint logic programming
 - language 282
- contradiction 14
- contradictory branch, of a semantic tableau 34
- contradictory branch, of a tree 145
- contradictory tableau 34, 142
- contrapositive law 20
- contrapositive proof 59
- crime solving 294–295
- cut 240–245, 284
- cut-failure combination 272
- data** 204–205, 210, 215
- data, addition 252–253
- data, deletion 253
- data, ill-defined 218, 242
- data, lists 228
- data, management queries 220
- data, syntax 222
- data, verification queries 220
- database 216, 219, 276

- decision problem 168–169
- decision procedure 168
- declarations, program 217
- declarative interpretation
 - of a program 235–238
- deduction, Beth 60
- deduction, compactness
 - theorem 64
- deduction, completeness
 - theorem 62
- deduction, soundness theorem 61
- deduction, theorem of, predicate
 - logic 110
- deduction, theorem of, propositional
 - logic 40
- definition, recursive 245
- degree, of a predicate 96
- Δ -PROLOG 282
- De Morgan's laws 20, 44
- dense order 124
- depth-first search strategy 238–240
- derivation, from axioms 43
- descendant 63
- descendant, immediate 63
- descriptive language 210
- deterministic nature of PROLOG 238
- diagnosis system 278–281
- disagreement set 155–156
- disjunction 2, 6, 46
- disjunctive normal form,
 - propositional logic 29
- distributivity 44, 106–107, 126–127
- double negation law 20
- dragon 198–200
- dual, to a quantifier 97

- efficiency of a program 243
- elementary extension,
 - of an interpretation 121
- elementary extension,
 - of a language 120
- empty clause 46
- empty list 229
- empty set of clauses 47
- empty substitution 100
- Epicurus 203
- equal by definition 13
- equality 108, 255–256
- equivalence 3, 6, 15
- equivalences, substitution of 39, 106
- excluded middle 14, 20

- existential quantifiers 96
- expert systems 275
- extension, of a truth valuation 12
- extension, elementary,
 - of an interpretation 121
- extension, elementary,
 - of a language 120
- extension, non-elementary,
 - of an interpretation 121
- extension, non-elementary,
 - of a language 121

- fact, entering a new 252
- fact, expert system 271
- fact, predicate logic 112
- fact, program 216–217
- fact, propositional logic 48
- fail, for a goal 48, 167, 213
- failure-cut combination 272
- false, logically 14
- false, sentence 117–118
- family tree 244–248
- final node 144
- final state 238–239
- finding a path 239
- finite degree, tableau 63
- first step 6
- flow, of a program 212
- FORTTRAN 204, 209
- FORTTRAN program 14
- formula, closed 100
- formula, matrix of 126
- formula, predicate logic 94, 97, 106
- formula, prefix of 126
- formula, prenex 126
- free occurrences, of variables 99
- free variable 99
- free variable, for a term
 - in a formula 103
- Frege 42
- full-stop symbol 206, 215–216, 229–231
- functional programming 2, 214
- functions, PROLOG 222, 225
- functions, symbols of 96
- functor 230

- general programs 268
- general unifier 155, 232–235
- general unifier, most 156
- generalized predicate 230

- generalization rule, predicate logic 104
- geometry, Euclidean 192–195, 200–201
- Gentzen 31
- goal, definite 48
- goal, fail 167, 213
- goal, normal 264
- goal, predicate logic 111
- goal, program 48
- goal, PROLOG 216
- goal, propositional logic 48
- goal, succeed 48, 167
- Goedel 167
- ground instance, of a clause 111
- ground term 99
- group 184–185
- guarded Horn clauses 284

- H**anoi, towers of 260–261
- head, of a clause 48
- head, of a list 229, 231, 248
- head, of a rule 216, 218
- Heraklith 93
- Herbrand interpretation 131, 134, 150
- Herbrand theorem 135, 149
- Herbrand universe 132
- heuristics 276
- high level symbol processing languages 277
- Hilbert 18
- Hintikka's lemma 56–57
- Horn clause, guarded 276
- Horn clause, predicate logic 111, 204–205, 215
- Horn clause, propositional logic 48

- IC-PROLOG 282
- if and only if 6
- if/then 6
- immediate descendant 63
- imperative commands 210
- implication 3, 6
- incomplete tableau 34
- inconsistent, semantically 23
- inconsistent, sentence 122
- inconsistent, set of sentences 122
- indirect proof 59
- induction, complete 7
- induction, course of values 7
- induction, principle of mathematical 6–7
- inductive scheme for tableaux 55
- inductive step 6
- inequality, PROLOG 256
- inference mechanism 204, 236, 276
- infix operators 258
- initial state 238–239
- instance, ground, of a clause 111
- instantiation of variables 154, 206, 232
- interaction with program 253–255
- interface with system 277
- intermediate state 238
- internal representation of a list 230
- interpretation, agrees with branch 164
- interpretation, elementary extension of 121
- interpretation, declarative 217
- interpretation, Herbrand 131, 134, 150
- interpretation, predicate logic 95, 114–116
- interpretation, procedural 217
- interpretation, propositional logic 24
- interpretation, theory of the 120
- intuitionistic logic 18

- keyboard interaction 253
- kidney disease system 278
- knowledge aquisition systems 277
- knowledge based systems 276
- knowledge engineer 277
- knowledge management 276
- knowledge representation 276
- Koenig's lemma 63
- Kripke semantics 14

- λ -calculus 214
- language, arithmetic 97, 134
- language, descriptive 210
- language, elementary extension of 120
- language, imperative 210
- language, predicate logic 96
- language, programming 204
- language, propositional logic 6
- language, relational 276
- law, contrapositive 20
- law, De Morgan 20
- law, double negation 20
- law, excluded middle 20

- law, Pierce 35
- law, syllogistic 20
- law, transportation 20
- linear resolution 213
- LISP 228, 277
- list, body 229
- list, concatenation 249–251
- list, empty 229
- list, head 229, 231, 248
- list, management 248
- list, non-empty 229
- list, PROLOG 228–231
- list, subsets 249
- list, tail 229, 231, 248
- list, tree structure 231
- list, unification 248
- literal, predicate logic 110
- literal, propositional logic 44
- logic, interpretation of a program 235
- logic, intuitionistic 18
- logic, modal 14
- logic, nonmonotonic 273–275
- logic, predicate 93
- logic programming 31, 131, 163, 203–204, 214
- logic programming, constraint 282
- logic, propositional 5
- logical connectives 6, 10, 96
- logical length, of a proposition 6
- logically equivalent 15
- logically false 14
- logically true 14
- logically true, formula 124
- loops, program 242, 246

- management**, data, predicates 252
- management**, data, queries 220
- management**, knowledge 268
- management**, lists 248
- marginal conditions**, for a relation 247
- matching** 232–235
- matrix** of a formula 126
- maximal consistent set** 77
- metalanguage** 21
- metaprogramming** 283
- metaproposition** 21
- MICRO-PROLOG** 282
- middle**, excluded 14, 20
- modal logic** 14
- modal operators** 14
- model**, of a sentence 120
- modus ponens** 38, 42, 104
- Morgan**, De 20
- most general unifier** 156
- mu*-PROLOG** 282
- murder problem** 294–295

- neck symbol** 47–48, 215
- negation** 2, 6
- negation**, by failure 263, 264
- negation** by failure, rule 264
- negation**, double 20
- negation**, PROLOG 262
- negation**, safe 267
- negation**, unsafe 267
- negative knowledge** 263, 270
- next node** 143
- node**, final 144
- node**, next 143
- node**, previous 143
- node**, tableau 34
- node**, tree 143
- non-elementary extension**, of an interpretation 121
- non-elementary extension**, of a language 121
- non-empty list** 229
- nonmonotonic logic** 273–275
- non-satisfiable**, proposition 23
- non-satisfiable**, sentence 122
- non-satisfiable**, set of sentences 122
- non-verifiable**, clause 46
- non-verifiable**, proposition 14, 23
- non-verifiable**, sentence 122
- non-verifiable**, set of sentences 122
- normal forms** 27
- normal forms**, conjunctive, propositional logic 29
- normal forms**, disjunctive, propositional logic 29
- normal forms**, prenex 125
- normal forms**, Skolem 125, 129
- normal**, goal 264
- normal**, program 268
- normalization** of variables 160
- not** 6
- numbers**, PROLOG 222, 224
- nu*-PROLOG** 282

- object-oriented programming** 214

- objects, PROLOG 222
- objects, type checking 257
- OCCAM 283
- occur check 157
- occurrences, bound, of variables 99
- occurrences, free, of variables 99
- operand position 258
- operand priority 259
- operation mechanism, PROLOG 232
- operators, associativity 259
- operators, infix 258
- operators, modal 14
- operators, position 258
- operators, postfix 258
- operators, prefix 258
- operators, priority 257–258
- operators, PROLOG 257–260
- or 6
- origin, of a tree 143

- parallelism 283–284
- PARLOG 284
- parentheses 6, 96
- partial validity 95
- PASCAL 204, 209
- path, finding 239
- path, solution 239
- period symbol 206, 215–216, 229–231
- Pierce's law 35
- Pompei 205–209
- position of an operator 258
- postfix operators 258
- predicate logic 95
- predicate logic, axioms 104
- predicate, arity of 96
- predicate, compound 225
- predicate, data management 252–253
- predicate, degree of 96
- predicate, generalized 230
- predicate, symbols 95–96
- predicate, tree structure 226–228
- predicates 94, 225–226
- predicates, built-in 252
- predicator 229
- prefix operators 258
- prefix, of a formula 126
- prenex normal form 125
- prenex formula 126
- previous node 143

- priority for operands 259
- priority for operators 257–258
- problem, decision 168–169
- procedural interpretation 217
- procedural part, of a program 216, 235
- procedure, decision 168
- procedure, program 217
- program 112, 209
- program, data 215
- program, declarative interpretation 235–238
- program, efficiency 243
- program, flow 212, 252
- program, interaction 253–255
- program, logic interpretation 235–238
- program, loops 242, 246
- program, normal 268
- program, procedural interpretation 235–238
- program, queries 215
- program, stratified 269
- program, structure 215
- program, updating 252
- programming, functional 2, 214
- programming, constraint logic 282
- programming, language 204
- programming, logic 31, 131, 163, 203–204
- programming, object-oriented 214
- PROLOG 43, 160, 163, 203–211, 214–215
- PROLOG, alphabet 222
- PROLOG, atoms 222, 224
- PROLOG, constants 222, 224–226
- PROLOG, deterministic nature 238
- PROLOG, dialects 282
- PROLOG, functions 222, 225–226
- PROLOG, lists 228
- PROLOG, negation 262
- PROLOG, numbers 222, 224
- PROLOG, operation mechanism 232
- PROLOG, predicates 225–226
- PROLOG, pure 285
- PROLOG, real 285
- PROLOG, special characters 222, 224
- PROLOG, through PROLOG 240
- PROLOG, tree structures 226–228
- PROLOG, variables 222–223
- PROLOG I 282

- PROLOG II 282
- PROLOG III 224, 282
- proof 40
- proof, axiomatic, predicate logic 109
- proof, axiomatic, propositional logic 38
- proof, Beth 36
- proof, contrapositive 59
- proof, indirect 59
- proof, resolution, predicate logic 160
- proof, resolution, propositional logic 31
- proof, systematic tableaux 136
- property, of a proposition 7
- propositional logic 5
- propositional logic, axioms 38, 42
- propositions, atomic 6
- propositions, compound 6
- Protagoras 1
- provable, Beth 36, 42, 142, 164
- provable, by resolution 52
- provable, from a set of formulae 109
- provable, predicate logic 109
- provable, propositional logic 40
- proving, automatic theorem 31, 131
- pure PROLOG 285

- quantifiers 95
- quantifiers, existential 96
- quantifier, dual to 97
- quantifiers, universal 96
- queries, complex 220
- queries, conjunctive 220
- queries, data management 220
- queries, data verification 220
- queries, of a program 215–216, 219–222

- read-only variables 284
- real PROLOG 285
- recursion, bound of 248
- recursive definitions 229, 240, 244–248, 251, 260, 296–298, 300
- recursive nature of unification 233
- recursive relations 245
- reflexivity 108
- refutable, Beth 36, 142
- refutable, by semantic tree 146
- refutation, Beth 36
- relational language 284
- relations 210
- relations, boundary conditions 247
- relations, marginal conditions 247
- relations, *n*-ary 116
- relations, recursive 240, 245
- renaming substitution 103
- resolution, completeness theorem 65–66, 165
- resolution, linear 213
- resolution, NF 264
- resolution, predicate logic 153, 160–163
- resolution proof, predicate logic 160
- resolution proof, PROLOG 204–209, 232
- resolution proof, propositional logic 31, 43, 48, 52
- resolution, rule of 49–50
- resolution, soundness theorem 65, 165
- resolvent, of clauses 50
- Robinson 131, 158
- rule, of resolution 49–50
- rules 42, 216, 218, 268
- rules, entering new 253

- safe, negation 267
- satisfiable, proposition 14, 23
- satisfiable, sentence 122
- satisfiable, set of sentences 122, 165
- satisfiable, sequence 62
- satisfy, truth valuation, of sequence 62
- screen interaction 254
- selection function 213
- semantic tableau 31, 34
- semantic tableaux, completeness theorem 59
- semantic tableaux, soundness theorem 58
- semantic tree 144
- semantic tree, refutable by 146
- semantically consistent 23
- semantics, Kripke 14
- semantics, propositional logic 5–6, 10
- sentence 100, 112
- sentence, consistent 122
- sentence, false 117–118
- sentence, satisfiable 122
- sentence, true 117–118
- sentence, universal 128
- sentence, verifiable 122
- sequences of special characters, PROLOG 222, 224
- set, disagreement 155–156
- set, of sentences, consistent 122
- set, of sentences, inconsistent 122

- set, of sentences, non-satisfiable 122
- set, of sentences, non-verifiable 122
- set, of sentences, satisfiable 122
- set, of sentences, verifiable 122
- set-theoretic representation, of
 - a proposition 46
- set-theoretic representation, of
 - a sentence 130-131, 166, 205
- shells 277
- short truth table 19
- signed formula 32
- Skolem function 129
- Skolem normal form 125, 129
- solution path 239
- soundness theorem, axiomatic
 - proofs, predicate logic 109, 119
- soundness theorem, axiomatic
 - proofs, propositional logic 64
- soundness theorem, deductions 61
- soundness theorem, NF 267, 269
- soundness theorem, resolution
 - 65, 165
- soundness theorem,
 - semantic tableaux 58
- soundness theorem,
 - tableaux proofs 164
- special characters, PROLOG 222, 224
- state, final 238
- state, initial 238
- state, intermediate 238
- state space 238, 247-248, 250
- stratification 268
- stratified, program 269
- structure, program 215
- structures, from predicates 225
- structures, tree, list 231
- structures, tree, PROLOG 226-228, 232-235
- subform(A) 9
- subformula 9, 98
- subgoal, predicate logic 111
- subgoal, propositional logic 48
- subterm 98
- substitution 100, 204-209, 232-235
- substitution, empty 100
- substitution of equivalences 39, 44, 106
- substitution, renaming 103
- substitution set 100
- substitutions, composition of 101
- succeed, for a goal 48, 167
- syllogistic laws 20
- symbols, predicate logic 95-96
- symbols, propositional logic 6
- symmetry 108
- syntactic analysis 226-228
- syntax, data 222
- syntax, predicate logic 96-97
- syntax, propositional logic 5-6
- system, automatic knowledge
 - aquisition 277
- system, knowledge based 276
- systematic tableau, complete 138, 141
- systematic tableaux proofs 136
- tableau, atomic 136-137
- tableau, complete 34
- tableau, complete systematic 138, 141
- tableau, contradictory 34, 142
- tableau, incomplete 34
- tableau, proof, compactness 165
- tableau, proof, completeness 164
- tableau, proof, soundness 164
- tableau, proof, systematic 136
- tableau, semantic 31, 34
- tables, truth 10-11, 17
- tail, of a clause 48
- tail, of a list 229, 231, 248
- tail, of a rule 216, 218
- tautology 14, 22, 44, 105
- term, atomic 224
- term, ground 99
- term, predicate logic 97
- theorem proving, automatic 31, 131, 214
- theorems 42
- theory, of the interpretation 120
- theory, of types 214
- thief 236-238
- towers of Hanoi 260-261
- transition, allowed 238
- transitivity 108
- transportation law 20
- trapezium 200-201, 288
- tree 143
- tree, refutable by semantic 146
- tree, semantic 145
- tree, state space 247-248, 250
- tree, structures, PROLOG 226-228, 232-235

- tree, structures, list 231
- true, logically 14
- true, logically, formula 124
- true, sentence 117–118
- truth table 10–11, 17
- truth table, short 19
- truth valuation 11
- truth valuation, agrees with branch 56
- Turing 131
- TURBO-PROLOG 282, 274
- type checking 257
- types, theory of 214

- unification algorithm** 154–155, 157–158, 232–235
- unification, lists 248
- unification, predicate logic 153, 204–209
- unification, process 223, 228, 232–235
- unifier 156
- unifier, general 155, 232–235
- unifier, most general 156
- unit clause 48
- universal quantifiers 96
- universal sentence 128
- universal validity 95
- universe, Herbrand 132
- universe, of an interpretation 116

- unsafe, negation 267
- unused, node 34
- updating variables 232
- used, node 34

- validity, partial** 95
- validity, universal 95
- valuation, Boolean 11
- valuation 10
- valuation, truth 11
- variables 94, 97, 223
- variables, anonymous 223
- variables, bound 99
- variables, bound occurrences 99
- variables, free 99
- variables, free for a term in a formula 103
- variables, free occurrences 99
- variables, normalization 160
- variables, PROLOG 223
- variables, read-only 284
- variables, updating 232
- variants, of sets of formulae 103
- verifiable, proposition 14, 23
- verifiable, sentence 122, 132
- verifiable, set of sentences 122
- verification queries, data 220

- warehouse problem** 290–292
- well-formed expression 6